De Montfort University

A Parallel Transformations Framework for Cluster Environments

by

Peer Bartels

A thesis submitted in partial fulfilment for the
degree of Doctor of Philosophy

in the
Faculty of Technology
Software Technology Research Laboratory

September 2011

Declaration of Authorship

I, Peer Bartels, declare that this thesis titled, "A Parallel Transformations Framework for Cluster Environments" and the work presented in it are my own and has been produced by me as the result of my own original research. I further confirm that:

- This work was done wholly while in candidature for a research degree at this University during the period of October 2006 to September 2011;
- It has not been submitted, either wholly or substantially, for another Honour School or degree of this University, or for a degree at any other institution;
- Where I have consulted the published work of others, this is always clearly attributed;
- I have clearly signalled the presence of quoted or paraphrased material and referenced all sources. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- I have not sought assistance from any professional agency;
- Either none of this work has been published before submission, or parts of this work have been published except the ones mentioned in Section: 2

I agree to retain an electronic copy of this work until the publication of my final examination result, except where submission in hand-written format is permitted. I agree to make any such electronic copy available to the examiners should it be necessary to confirm my word count or to check for plagiarism.

Signed: _____

Date: _____

Publications

M. Alfawair, O. Aldabbas, P. Bartels, H. Zedan. Grid Evolution. 2007 International Conference on Computer Engineering and Systems, ICCES'07. pp. 158-163.

P. Bartels. Clustering Techniques on Transformation Systems. In Proceedings of Informatiktage (GI, 2008), Volume S-6 of LNI, pp. 61-64 (2008).

De Montfort University

Abstract

Faculty of Technology
Software Technology Research Laboratory

Doctor of Philosophy

In recent years program transformation technology has matured into a practical solution for many software reengineering and migration tasks.

FermaT, an industrial strength program transformation system, has demonstrated that legacy systems can be successfully transformed into efficient and maintainable structured C or COBOL code. Its core, a transformation engine, is based on mathematically proven program transformations and ensures that transformed programs are semantically equivalent to its original state. Its engine facilitates a Wide Spectrum Language (WSL), with low-level as well as high-level constructs, to capture as much information as possible during transformation steps. FermaT's methodology and technique lack in provision of concurrent migration and analysis. This provision is crucial if the transformation process is to be further automated. As the constraint based program migration theory has demonstrated, it is inefficient and time consuming, trying to satisfy the enormous computation of the generated transformation sequence search-space and its constraints.

With the objective to solve the above problems and to extend the operating range of the FermaT transformation system, this thesis proposes a Parallel Transformations Framework which makes parallel transformations processing within the FermaT environment not only possible but also beneficial for its migration process. During a migration process, many thousands of program transformations have to be applied. For example a 1 million line of assembler to C migration takes over 21 hours to be processed on a single PC. Various approaches of search, prediction techniques and a constraint-based approach to address the presented issues already exist but they solve them unsatisfactorily. To remedy this situation, this dissertation proposes a framework to extend transformation processing systems with parallel processing capabilities. The parallel system can analyse specified parallel transformation tasks and produce appropriate parallel transformations processing outlines. To underpin an automated objective, a formal language is introduced. This language can be utilised to describe and outline parallel transformation tasks whereas parallel processing constraints underpin the parallel objective.

This thesis addresses and explains how transformation processing steps can be automatically parallelised within a reengineering domain. It presents search and prediction tactics within this field. The decomposition and parallelisation of transformation sequence search-spaces is outlined. At the end, the presented work is evaluated on practical case studies, to demonstrate different parallel transformations processing techniques and conclusions are drawn.

Acknowledgements

I would like to acknowledge the following people who helped me in many different ways when I undertook the work of this PhD thesis.

My deepest gratitude goes to my supervisor, Professor Hussein Zedan, Head of the Software Technology Research Laboratory (STRL) at the De Montfort University, Leicester, United Kingdom. His broad knowledge and his logical way of thinking has been of real value to me. His understanding, commitment and personal guidance and contributions have helped me during my three year PhD study.

I also want to thank my supervisor, Dr. Martin Ward, for providing me with his detailed and constructive comments, including his helpful suggestions during the many discussions we had, which gave this thesis a special touch.

I also want to express my greatest gratitude to Prof. Dr. Karl Hajo Siemsen. Without his help and commitment to the partnership between the STRL and the Hochschule Emden/Leer and his encouragement to pursue my PhD degree, this thesis would not exist.

I would also like to give my thanks to the Research Office at the De Montfort University for their outstanding support and management during this study, including the always friendly and very helpful staff of the De Montfort University.

I would like to thank all colleagues in the Software Technology Research Laboratory (STRL) at the De Montfort University for their outstanding support, valuable suggestions, their encouragement and many discussions: Stefan Natelberg, Matthias Ladkau, Keno Buss, Sascha Westendorf, Shaoyun Li, Feng Chen and many others.

Finally, I also want to express many, many thanks to my parents and my sister for all their love, encouragement, patience and support over the past years. This thesis has been dedicated to them.

Contents

List of Figures

Listings

List of Tables

List of Acronyms

Ada	A structured, typed, imperative, wide-spectrum and object-oriented high-level programming language
ADL	Architecture Description Language
AIX	Advanced Interactive eXecutive
ANSI	American National Standards Institute
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AST	Abstract Syntax Tree
Beowulf	Beowulf is a computer cluster class of Commercial Off-The-Shelf (COTS) components, first introduced by NASA
BNF	Backus-Naur Form
C	Programming Language C
CC	Cyclomatic Complexity
C++	Object-Oriented Programming Language C
CASE	Computer Aided Software Engineering
CLI	Command Line Interface
COBOL	Common Business-Oriented Language
COTS	Commercial Off-The-Shelf
CBPTT	Constraint Based Program Transformation Theory
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
DMS	Design Maintenance System
DNS	Domain Name Service
DSSA	Domain Specific Software Architecture

DynDNS	Dynamic Domain Name Service
FCE	FermaT Cluster Environment
FTE	FermaT Transformation Engine
FME	FermaT Maintenance Environment
Fortran	Formula Translation/Translator
FPGAs	Field-Programmable Gate Arrays
GA	Genetic Algorithms
GCM	Global Cluster Manager
GUI	Graphical User Interface
GCA	Grand Challenge Application
HA	High-Availability
HDD	Hard Disk Drive
HPC	High Performance Cluster
HP-UX	Hewlett Packard UniX
HTC	Honeywell Technology Center
IBM	International Business Machines
IBM 360	IBM System/360 (S/360) is a mainframe computer system family announced by IBM
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
Java	A general purpose, high-level, object-oriented, cross-platform programming language developed by Sun Microsystems
JNI	Java Native Interface
JOVIAL	A high-order computer programming language, specialized for the development of embedded system
LAN	Local Area Network
Linux	Is a generic term referring to Unix-like computer operating systems based on the Linux kernel
LISP	LISt Processing
LOC	Lines Of Code
MIB	Management Information Base
MICOM	Missile Command

MPI	Message-Passing-Interface
MS	Microsoft
NFS	Network File System
NASA	National Aeronautics and Space Administration
NoCC	Number of Code Characters
OS	Operating System
PC	Personal Computer
Perl	A high-level general-purpose Unix scripting language
PostgreSQL	Open source object-relational database system
PTTD	Parallel Transformation Task Description
PTTDL	Parallel Transformation Task Description Language
PVM	Parallel Virtual Machine
RAM	Random Access Memory
RMI	Remote Method Invocation
RPC	Remote Procedure Call
Scheme	Scheme is one of the two main dialects of the programming language LISt Processing (LISP)
SETI	Search for Extraterrestrial Intelligence
SED	Software Engineering Directorate
SMP	Symmetric-Multi-Processing
SNMP	Simple Network Management Protocol
SSH	Secure Shell
Sun	Sun Microsystems is a multinational vendor of computers, computer components, computer software, and information technology services
STRL	Software Technology Research Laboratory
TCP	Transmission Control Protocol
TTD	Transformation Task Description
TSDL	Transformation Scheme Description Language
Ubuntu	Computer operating system based on the Debian GNU/Linux distribution
UML	Unified Modelling Language

UNICODE Computing industry standard for the representation and manipulation of text

UNIX Computer operating system developed in 1969 by a group of AT&T at Bell Labs

VCS Vertias Cluster Server

VHDL Hardware description language used to describe digital and mixed-signal systems such as FPGAs

Windows A series of software operating systems and graphical user interfaces produced by Microsoft (MS)

WLAN Wireless Local Area Network

WSL Wide Spectrum Language

XML Extensible Markup Language

"To my parents, my sister and my second family for their love and support."

Chapter 1

Introduction

Objectives

- To express the need for a parallel transformations framework.
- To present the scope of the thesis.
- To define the respective research questions.
- To highlight the original contributions.
- To give a brief overview of the thesis structure.

1.1 Motivation and Targets of the Presented Research

There is a vast collection of operational software systems worldwide which are crucial to users, yet these systems are becoming increasingly difficult to maintain, to enhance and to keep up to date with the rapidly changing requirements. It does not seem economically viable to replace or to review these low-level legacy systems. This is particularly the case as legacy assembler systems have high maintenance costs, and migration of such systems to a different environment is of high complexity, compared to the migration of high-level systems.

Software maintenance is the highest cost factor during software life cycle, usually consuming between 50% to 90% of the project total budget. One of the reasons is that

the work of the maintainer is still hampered by lack of maintenance tools. Much of their work involves code analysis, whether it is finding an obscure bug, attempting to understand a piece of code prior to modification, software enhancement or analysing the possible effects of a source code modification.

Program transformation technology has proven to be one solution to deal with this situation. It has matured into a practical solution for many software reengineering and migration tasks. The FermaT transformation system is one example of transformation technology which utilises formal proven program transformations. The uniqueness of these transformations is, they preserve or refine the semantics of a program while changing its form [1]. The transformations can be utilised to restructure, to simplify or to extract high-level representations of the legacy systems. Through the use of an appropriate sequence of transformations, the extracted representation is guaranteed to be equivalent to its original code logic. This method is based on a Wide Spectrum Language (WSL) accompanied by formal methods. Over the last sixteen years FermaT has developed into a large system with a huge catalogue of proven program transformations, which have been applied to many software development, reverse engineering and maintenance problems [2]. These program transformations are applied to restructure the Wide Spectrum Language (WSL) code in a semi-automated manner, through static maintainer knowledge based scripts.

Most of todays reengineering systems lack in parallel processing provision, as they do not support parallel reengineering [3]. Recent advances in these technologies have led to the availability of inexpensive clusters computers, consisting of Commercial Off-The-Shelf (COTS) components such as networks of computers (PCs, workstations, SMPs) [3]. These systems provide an appealing vehicle for cost-effective parallel computing, and play a major role in todays cluster computing domain [4]. The approach proposed in this thesis enlarges the FermaT transformation system [2] by providing an environment to perform parallel transformation tasks and complex parallel transformations computations on COTS parallel processing components. The utilised and developed techniques introduce an automated parallel transformations processing approach to the FermaT environment. To support a parallel processing behaviour, a formal language is introduced to describe and outline parallel transformations processes. Speed-up of program transformation processes can be achieved through the usage of parallel processing constraints. By the provision of these features, the maintainer can specify parallel transformations processing roadmaps and directly assign transformation tasks to computing nodes.

This research aims to provide an insight into the nature of parallel transformations computing, while proposing a parallel transformations framework to the reengineering

domain. The utilised and specified techniques are presented and discussed within this thesis.

1.2 Scope of Thesis

This thesis illustrates a new approach of parallel transformations processing within the software migration domain. It describes the establishment of a cluster based automated parallel transformations processing environment, utilising FermaT's transformation engine and its underpinning program transformation theory [5]. It outlines a new, dynamic and structured facility in which parallel transformations processing can be achieved. The basis for this technique is the development of a formal language to express and describe the behaviour of parallel transformations processes. A specified analysing system evaluates transformation tasks. On the basis of this system, the clusters headnode computes and produces suitable parallel transformations processing outlines. A transformation scheme description [6] decomposition technique assists the parallelisation process. This thesis concentrates on the motivation, description and realisation of a parallel transformations processing framework. Its scope includes the following content:

1. Analysis of the FermaT transformation system and identification of techniques for the realisation of parallelism.
2. Identification of the key features for the cluster based parallel transformations processing framework.
3. Evaluation of techniques of Wide Spectrum Language (WSL) code analysis.
4. Definition of algorithms for the decomposition of transformation scheme descriptions.
5. Analysis of parallel processing techniques to obtain an automated transformations processing environment.
6. Identification of search and prediction tactics for parallel transformations processing.
7. Analysis and development of a communication system for parallel transformations processing.
8. Implementation of the presented parallel transformations processing framework by utilising the FermaT transformation engine and its theory.
9. Presentation and implementation of the FermaT Cluster Environment (FCE), a prototype tool to demonstrate the presented approach.

1.3 Research Questions

To give a direction for the investigation a question was formulated. This research was driven by a number of challenges, introducing a parallel transformations processing framework for software migration processes. This thesis tries to answer the following overall research question:

"Can the transformation process in reengineering of systems be automated?"

In order to answer the question a number of sub-questions were formulated:

- How can automated parallel transformations processing be achieved?
- Which techniques can be utilised to decompose transformation scheme descriptions?
- Can transformation search problems be mapped to a parallel computing environment?
- How well can parallelisation be integrated within the FermaT transformation system?
- How big are the advantages of a parallel program transformations approach against a common one?

1.4 Original Contributions

The original contributions can be summarised as follows:

- The most significant contribution is the development of a parallel transformations processing system. The fundamental part for this automated approach is the definition of a formal language, utilised to express and outline the behaviour of parallel transformation tasks. The language is equipped with capabilities to define and directly assign parallel transformation tasks to computing nodes.
- The second contribution is the development of a parallel transformation task analysing system. The developed framework functions as a pre-processing environment. Major information about parallel transformation tasks are extracted and analysed for the development of parallel computation models. Additional task specific parameter are evaluated, WSL program source, speed of the parallel processing system and task specific constraints.

- The third contribution is the establishment of a transformation scheme decomposition technique. The developed methods and laws are utilised to decompose and map generated transformation sequence search spaces to the parallel transformations processing environment.
- The forth contribution is the refinement of parallel processing techniques for parallel computation of transformation processes. This includes their scheduling- and communication-techniques.
- The fifth contribution is the development and implementation of supporting tools to demonstrate the applicability, scalability and persistency of the presented parallel approach.

1.5 Success Criteria

The success of the presented approach can be validated, if both parallel transformations processing model and their supporting algorithms resolve the research question. The presented case studies and the demonstration of the prototype tool should reveal that the output results match the results obtained by manual calculation.

Success within the constraint based program transformation domain can be measured by the satisfaction or none satisfaction of the reengineering constraints "C_n" embedded within a transformation scheme description [6]. To assist parallel transformations process computation, the proposed parallel transformations processing language can be utilised to specify and guide parallel transformation tasks. The definition and the embedding of parallel processing constraints used to accelerate transformation processes should not limit or restrain the satisfaction of reengineering aims. This is only due to the fact that task specific parallel computation refinements do not have any influence on the defined reengineering constraints "C_n" or vice versa. This is also indispensable because both constraint categorisations need to be considered as independent to achieve parallelism.

To ensure that parallel transformation tasks are computed by the proposed parallel environment, they should have no restrictions. Nonetheless the maintainer is always notified beforehand if a specified parallel transformation task can be satisfied according to its refinements. Thus the successful computation of reengineering aims can only be evaluated during its execution.

1.6 Organisation of Thesis

The presented thesis is structured as follows:

- ***Chapter 1*** defines the research objectives, explains the research characteristics, selects the research method, identifies the research questions, highlights original contributions and defines the success criteria of the presented approach.
- ***Chapter 2*** provides an overview of theories and techniques for software reengineering, legacy systems and migration. Parallel computing techniques, modern cluster-systems, including their job-scheduling- and distribution-algorithms, are also discussed.
- ***Chapter 3*** highlights theory and program transformation techniques of the FermaT transformation system. How constraints can be utilised to satisfy reengineering aims within program transformation theory is also discussed.
- ***Chapter 4*** outlines the proposed parallel transformations processing framework, the needs for such a system and the technical steps for the realisation of this approach.
- ***Chapter 5*** describes the realisation of the proposed parallel transformations processing framework, the services which are needed to fulfill parallel transformations
- ***Chapter 6*** presents decomposition techniques and laws developed to decompose transformation scheme descriptions.
- ***Chapter 7*** discusses the utilised and refined parallel computing techniques.
- ***Chapter 8*** presents the implementation and framework of the FCE supporting tool.
- ***Chapter 9*** presents different case studies on WSL programs, and demonstrates the value of the presented parallel approach.
- ***Chapter 10*** summarises this thesis and draws conclusions of the presented work. It also replies to the research questions, addresses future work and related projects.

Chapter 2

Background and Related Research

Objectives

- To present the basic concepts of software reengineering and program transformation.
- To discuss legacy systems and the latest program transformation migration approaches.
- To provide an overview of parallel program transformation automation and migration environments.
- To state the methodology of parallel computation.
- To present an overview of today's most common parallel computation systems. techniques

2.1 Introduction

This research aims to provide a parallel transformations processing framework on the basis of the FermaT transformation system [1]. The framework is focused on four objectives: (1) Utilisation of a program transformation system for reengineering program

sources; (2) Application of program transformations stored in the transformation bank; (3) Analysis of legacy systems, transformation scheme descriptions and reeningeering aims; (4) task distribution to a parallel architecture. This chapter reviews techniques related to these areas.

2.2 Software Engineering

Since the early days of computer science in the 1940s, applications and the use of computers has grown at a breathtaking rate. In today's world, software plays a central role in domains of: government, banking and finance, education, transportation, entertainment, medicine, agriculture, and law. As a result, its number, size and application domain has grown rapidly. Billions of dollars are invested on development and maintenance of software. It can be said that the livelihood and lives of millions of people depend upon the success of software. In most aspects, software products have helped mankind to act more efficiently and more productively [7]. Today's software development leads to serious problems in costs, timeliness and quality of many software products. There are many reasons for this [8]:

- Software products are among the most complex "man-made systems". Software represents characteristics of: complexity, invisibility, and changeability [9].
- Programming techniques and processes that work most effectively for an individual person or a small team are often not a sufficient basis for the development of large and complex systems with millions of lines of code, requiring years of work, by hundreds of software developers.
- The speed of change in the computer and software technology domain drives the claim for new and evolved software products. This results in a situation which raises customer expectations and competitive forces to produce software within acceptable development circles.

The first organised formal discussion on the term of software engineering has taken place at a NATO Conference in 1968 [10]. Since then, the term software engineering has been widely used in areas of industry, government, academia and hundreds of thousands of computing professionals, who now go by the title software engineer. Numerous publications, groups, organisations, and professional conferences use the expression software engineering and many educational courses and programs on software engineering exist as well. However there are still disagreements about the meaning of the term itself.

2.2.1 Software Reengineering and Software Maintenance

Software engineering can be categorised into two sub-domains: software reengineering and software maintenance. The term software reengineering can be described as a process to improve or transform existing software, so that it can be understood, controlled, documented and used anew. Chikofsky and Cross express the reengineering process in their paper, "Reverse Engineering and Design Recovery: A Taxonomy" from 1990 as [11]:

> "Reengineering is an examination and alteration of a system to reconstitute it in a new form."

The American National Standards Institute (ANSI) and the Institute of Electrical and Electronics Engineers (IEEE) specify software maintenance as:

> "The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment", according to ANSI/IEEE Std 729-1983.

In simple terms, the reengineering process is a modification of a software system taking place after the system has been engineered, by adding new functions or correcting errors. The process typically includes a combination of other software evolution processes such as reverse-engineering, re-documentation, restructuring, translation, and forward engineering. The main objective of a reengineering process is to understand the existing software system (specification, design and implementation), to achieve a higher level of abstraction [11]. A common processing step is the re-implementation of the system in a high-level language, to achieve benefits of [12]:

- To increase the maintainability of the system.
- To achieve performance improvements.
- To increase the interoperability between systems.
- To decrease the personal dependency on low-level software engineers.

Some of the above objectives are closely related. Improving performance for example, is often done on cost of decreasing maintainability [13]. Moreover, system maintainers were usually not involved in designing a system. Usually reverse-engineering tools are

facilitated to understand the underlying system, to perform appropriate steps of software maintenance. In the reverse-engineering approach, the subject system is generally the starting point of expertise.

The potential to use Computer Aided Software Engineering (CASE) environments for software- and system-maintenance has increased their use within organisations. To fulfil the true potential of these CASE tools, the main reverse-engineering functions for maintaining software systems are included. An overview of today's reengineering systems is provided in Section 2.3.

2.2.2 Forward- and Reverse-Engineering and Restructuring

Forward- and *reverse-engineering* are two sub-disciplines within the reengineering domain and can be categorised as [11]:

Forward engineering is the traditional process of moving from high-level abstractions and logical implementation independent design to the physical implementation of the system. The term forward is commonly used where it is necessary to distinguish this process form reverse-engineering. Forward engineering follows a sequence of processing steps, starting from the requirements level through designing its implementation.

Reverse-engineering steps are usually executed to improve software products as well as to analyse software systems. Utilising this technique supports a basic understanding of the underlying system and its basis structure, but also helps to understand the system's design level, aid maintenance, strengthen enhancement or support replacement. Reverse-engineering main intentions can be summarised as [11]:

- Identification of the system's components and their interrelationship.
- Development of another form or high-level of abstraction representation of the existing system.

Reverse-engineering generally involves the extraction of design artifacts which can be utilised to build system abstractions which are less system dependent. This process can start from any level of the engineering life-cycle and can therefore be considered as a process of examination rather than building a new system. There are two sub-areas within this field, widely referred to as *Re-Documentation* and *Design Recovery* [11]:

- **Re-documentation** is the creation or revision of a semantically equivalent representation within the same relative abstraction level. They can be alternative views

of: data flow, data structure and control flow. Re-documentation is known as the oldest form of reverse engineering. Commonly used tools are pretty printers and diagram generators. Their key feature is to realise and visualise the relationships among program components and their interrelationship.

- **Design Recovery** is another subset of reverse-engineering, and assists to identify meaningful higher level of abstractions of the subject system, by adding domain knowledge, external information and deduction of fuzzy reasoning to its observations. Design recovery must be consequently reproduced with the help of knowledge, which includes the information required for a person to fully comprehend what the program or system does.

To summarise, reengineering includes *forward-* as well as *reverse-engineering*, whereas reverse-engineering is needed to achieve a more abstract description, usually followed by some form of forward-engineering or restructuring which may include some forms of new requirements not met in the original system.

Restructuring in this context is usually understood as the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system external behaviour (functionality and semantics). To give an example, the FermaT transformation system acts semantically preserving during its application of restructuring transformation processes [2]. A restructuring transformation is often an appearance of altering code to improve its structure in the traditional sense of structured design. For example, a code to code transformation which recasts a program from an unstructured "*spaghetti source code form*" to a structured form. Restructuring processes can be performed without knowing the program behaviour. This opens the opportunity, to case a set of "*if statements*" into a "*case structure*" or vice versa, without knowing the program's purpose or anything to do with the problem domain. Therefore restructuring can be regarded as a creation of new versions of the code without modification of what it does nor include new requirements. However, it may lead to a better understanding of the subject system for future code adjustments and software maintenance.

2.2.3 Key Objectives of Reverse-engineering

The purpose of reverse engineering is to increase the overall comprehensibly of the system for both maintenance and new development. Beyond the above mentioned explanations there can be considered six key objectives of this technology [11]:

- **Cope with complexity:** Since systems are increasing in complexity, automated approaches could lead to a better understanding by using reverse-engineering methods and tools to obtain knowledge of the underlying system.
- **Generate alternate views:** Graphical representations have long been accepted as comprehension aids. So, reverse-engineering is needed to understand the system with data flow- and control flow-diagrams. Structure charts and entity relation diagrams are used for system documentation.
- **Recover lost information:** The continuing evolution of large, long-lived systems results in information loss of system design. Modifications are usually not reflected in documentation and therefore design recovery is important to evaluate knowledge of the existing system which helps to create a understanding of the system.
- **Detect side effects:** Modifications can lead to unintended side effects in the system that involves its performance. Both forward- and reverse-engineering methods can help to avoid this problem before a system failure occurs.
- **Synthesize higher abstractions:** Synthesising a higher-level of abstraction of a system is a difficult task. The main purpose of this technique is to abstract as much information as possible. Expert systems are needed to create alternative views of the system including different abstraction levels.
- **Facilitate reuse:** Reverse-engineering methods can help to detect candidates for reusable software components. The above mentioned reengineering principles and techniques can help to understand and maintain the process of software development life cycle by detecting reusable software components.

It has been stated by Chikofsky and Cross in their paper "Reverse Engineering and Design Recovery: A Taxonomy" [11], the cost of understanding software systems can be greatly reduced by the reengineering process at a specific point within this software life-cycle. Both forward- and reverse-engineering techniques can contribute to decrease 50% to 90 % of the total software development life-cycle cost. By reusing software engineering technologies already implemented in today's software maintenance environments.

2.3 Overview of Program Transformation Approaches

Maintenance and reengineering of legacy systems is a challenging task [14]. Often only source code is available, while design or requirement documents are lost or have not been kept up to date with the current system implementation. This scenario applies

to many old business applications which run on mainframes. Most of them are written in COBOL or a low-level programming language. It is widely known that software maintenance can consume up to 75 % of the software system life cycle cost [15]. Poor system structure and missing documentation contribute to these numbers. To perform changes to a software system is a difficult task if only the source code is available. Only strict and disciplined utilisation of the outlined software engineering principles could have reduced these problems [16]. There is an urgent need for reengineering tools which assist in program understanding and restructuring. In order to fully understand legacy systems, it is essential to recover and document its software architecture. Only when this can be assured, can the system be re-designed and reimplemented successfully to the satisfaction of the new system design. The following sections present some of todays most common re-engineering tools.

2.3.1 Stratego/XT

Stratego/XT [17] combines framework, language and tools for the development of program transformation systems. Its aim is to support a wide range of program transformation development. The framework comprises the transformation language Stratego and XT, a collection of transformation tools. The tools are designed for reuse in other transformation systems. The specified language is based on the paradigm of rewriting under the control of programmable rewriting strategies, whereas the tools provide the facility to establish the transformation system infrastructure, its parsing and pretty printing techniques.

The main intention of Stratego and XT is a better productivity in the development of transformation systems through the use of high-level representations of programs, domain-specific languages for the development of parts of a transformation system, and generation of various aspects of a transformation system in an automatic way. System transformation rules can only be applied on programs which are written in the Stratego language. A transformation rule is this context is defined as [17]:

> "A rule which encodes a basic transformation step as a rewrite on an Abstract Syntax Tree (AST)."

Theses transformation rules are comparable to FermaT transformations [18]. The Stratego language supports a wide variety of program transformation application, based on the paradigm of rewriting strategies. A program transformation specification consists of a signature, a set of applicability rules and a strategy for applying it [19].

The rewriting rules are not the actual primitive action of applying program transformations, they can be broken down into more basic actions and are domain specific. This allows an independent development and separation of strategies and facilitates a more careful control over their applicability. This usually results in the separation of transformations and its generic strategies. This technique gives its user the possibility to encode a wide range of program transformation idioms in a flexible way. The Stratego/XT transformation system and tools wrap composition rules and strategies into a stand-alone, deployable component which can be called from the command-line or from any other tool aimed for transformation processing. However within the current development stage of Stratego/XT a parallel transformations processing version does not exist. So similar to the present FermaT transformation engine it does not provide parallel processing features nor automated evaluation techniques.

2.3.2 FermaT Transformation Engine

The FermaT transformation system is a powerful industrial-strength program transformation system based on a Wide Spectrum Language (WSL) language [1]. FermaT s focus lies on reengineering IBM Assembler 360 mainframe code. It has been successfully used in several major assembler to C and assembler to COBOL migration projects, involving the conversion of millions of lines of hand-written assembler code to efficient and maintainable C or COBOL code. Assembler modules are translated into the intermediate language WSL. This language is used to simply, restructure, maintain or raise the abstraction level of program code, by utilising WSL program transformations. After the restructuring process, the WSL code is transformed into the target language, usually C or COBOL . The difficulty lies in the WSL program transformations application because they are applied in a semi-automated way, one by one on a specific WSL program AST path. Within each processing step, the maintainer usually has to evaluate on which path a transformation should be applied. Usually after the application of a sequence of transformations, it becomes complicated to evaluate the result, because it has to be based on personal knowledge. This is also the case if more than one transformation sequence is applied on the same program source or state. The difficulty also lies in the enormous applicability of code transformations which commonly slows down the transformation process. A transformation process pre-processing technique could eliminate this problem. An analysing system which evaluates applicable transformation sequences together with reliable transformation search tactics could automate and speed up the whole transformation process.

2.4 Parallel Program Transformation Systems

If sequential applications are to benefit from parallel processing techniques they have to be parallelised. It could be that sequential programs may have a large amount of explorable parallelism, however it is difficult for automatic techniques to determine the best parallelisation strategy. This is because parallel processing platforms do not share common paradigms or programming languages. Machine characteristics may also differ widely, depending on whether a particular platform has shared memory or distributed memory, whether its a distributed processing environment or a parallel computing environment. Language extensions for parallelism also differ from platform to platform.

Each reengineering process may have to be repeated for each parallel processing platform. Therefore it seems that some manual effort is unpredictable to extend sequential programs with parallel processing capabilities. This also depends on the availability and quality of program design. This process mostly depends on two fields, the identification of opportunities for parallelism and source code restructuring to promote parallelism [20]. To reduce the difficulty of parallelisation, design information of legacy systems should be available at an appropriate level of abstraction to achieve the designated optimisation process. For the parallelisation of transformations at source code level, important information might be data-flow dependence relations between program modules or statements, communication volume at the subroutine level or at the parallel partitioning level of module size and complexity measures, partition size measures or call graphs. Parallelising sequential source code can be distinguished between by hand parallelisation or automatic parallelisation.

Many reengineering methodologies and Computer Aided Software Engineering (CASE) tools are available for forward engineering. However there is a lack in tools for reverse engineering. Parallel reverse-engineering or parallel reengineering tools are seldom. Some of todays reengineering tools already have parallelism implemented, though it has to be distinguished between tools capable of transforming sequential source code and parallelising it or parallelising the transformation process [20, 21, 22, 23].

2.4.1 DMS: A Parallel-Reengineering Tool

The Design Maintenance System (DMS) Software Reengineering Toolkit is a set of tools, developed for customising source code [24]. It is knowledge based and can automatically analyse, modify, translate or generate any type of software to produce different programming languages, markup languages, hardware description languages, design notations and data descriptions. The system is able to analyse languages of COBOL, C,

C++, Fortran 95/90/77, JOVIAL, VHDL etc. The system is driven by semantics and captures the programs system design. Its focus is to support irregular parallelism on shared memory multi-processors. DMS supports basic reengineering principles of:

- Full lexical analysis of the program source, able to read source files in ASCII or UNICODE format.
- Automatic construction of an Abstract Syntax Tree (AST) including its manipulation and rewriting strategies.
- Source to source code transforming engine.
- A code generation component, using data structures and results of analysers to choose appropriate target machine idioms.

To be able to support the above features, the DMS system consists of an extremely generalised compiler, parser, semantic analyser, program transformation engine and a pretty printer. Each system component is highly parameterised and allows changes, to take a wide variety of effects on the system such as input languages, change of analysis, change of transformations and change of output. DMS is also unusual in that way because it has been implemented in the parallel programming language PARLANSE [21]. This language uses symmetric multiprocessors techniques and provides DMS with parallel reengineering capabilities. However the system lacks in performance while processing and analysing thousands of files at the same time. PARLANSE and DMS is based on an old Microsoft (MS) Windows system and therefore has limitations in relation to the maximum number of processing units (24 CPUs). The DMS system utilises a generalised parser to transform and restructure programming languages, whereas FermaT employs its intermediate language [2]. An environment analysing system as it is proposed within this thesis, which evaluates parallel transformation tasks before they are processed, would suit DMS well. The capability to dynamically add more computing resources during system runtime, would surely absorb some parallel computation limitations.

2.4.2 ControlH: A Parallel Processing Platform

Todays missile software domain has become increasingly complex in its use of parallel and distributed systems [25]. The US Army Missile Command (MICOM) and Software Engineering Directorate (SED) have established a testing application in which concurrent software development and evolution could be facilitated. The main intention of this approach is the distribution and modification of parallel processes. To fulfill this aim,

the reengineering system utilises the SED missile software architecture, which consists of a set of tools developed by Honeywell Technology Center (HTC).

The Architecture: The Domain Specific Software Architecture (DSSA) is focused on missile domain and provides an object based architecture that lays the emphasis on roles that objects perform within the system. Tools developed by HTC were chosen by SED because of their emphasis on the generation of a real-time embedded application system. Its software components do not only offer the automation of portions of code development, they also produce Ada code for the executive that controls multiple processes executed on multiple processors. The Honeywell toolset consists of three tool components: **ControlH**, **MetaH** and **MetaDoME** [26]:

- **ControlH** produces Ada code for guidance, navigation and system control, and supports analysis of these systems within a module testing context.
- **MetaH** creates the multiprocessor executive, based on paradigm that combines hand produced Ada code and code generated from ControlH. It establishes an easy mechanism to modify process execution rates and provides a schedule analysis of the processes in the system.
- **MetaDoME, or DoME** (Domain Modeling Environment) is a meta CASE tool which offers a graphical programming interface for ControlH and MetaH. The graphs established within the DoME are translated into an Architecture Description Language (ADL), which is interpreted by MetaH and ControlH and used to generate Ada code.

The project concentrates on providing a multi process simulation environment that executes systems major functional components in parallel. It is utilised to refine the SED missile DSSA environment. This demonstrates how parallel simulation environments like MetaDoME can be utilised to successfully map tested programming techniques onto a distributed processing environment. Unfortunately this is a military software application system and not meant for public use. The information available on this system is very limited.

2.5 Search based Program Transformation

The program transformation processes presented and utilised within this approach are based on the FermaT transformation engine [27]. The engine is based on a Wide Spectrum Language (WSL) [1]. FermaT is able to reengineer IBM Assembler modules to

maintainable C or COBOL code. The transformation process is semi-automated and faces some problems within its program transformation search- and application-domain. Two approaches introduced by Deji Fatiregun, Mark Harman and Robert M. Hierons could assist the search for satisfactory and applicable transformation sequences, hill-climbing and genetic search [28].

2.5.1 Program Transformation as a Search Problem

To transform programs into another form, program transformations are utilised. A program transformation process within the FermaT transformation engine transforms the initial WSL program "P_0" into another program state "P_n". If program transformations are applied in a sequence, they are considered as a transformation sequence. As during its application, many program transformations can be utilised, there also exist various WSL program states (P_1 - P_n) on the same code level.

Program transformations within the FermaT transformation process are applied on the Abstract Syntax Tree (AST) of the WSL program source. Each WSL node is either of general-, specific- or a group- type. The WSL syntax on which the FermaT AST is specified can be found in Appendix B.7. A FermaT transformation can be applied on a number of different AST types within a program state "P_i". The applicability of a transformation is usually determined by the applicability condition of the specified WSL AST node. According to this, each transformation process contains a possible triplet search-space Ω (α, δ, ρ) of:

- (α) Applicable FermaT transformations.
- (δ) WSL program state "P_i".
- (ρ) AST node within the program state "P_i", on which the program transformation is applied.

The program transformation search-space can be also categorised as a transformation sequence, in which the initially specified transformations are applied on various places within the program state "P_i". The focus on this specification lies in the order in which the transformations are applied, to achieve the desired reengineering aim. Transformation search-space specified is usually enormously large. This is the greatest disadvantage of the presented transformation search-space approach and makes the search infeasible [28].

2.5.2 Hill-Climbing Search Tactic

The hill-climbing transformation search tactic is based on heuristic methods. Before the initial search tactic starts, a transformation sequence is randomly chosen. This transformation sequence is the starting point. A specified fitness function evaluates the selected transformation sequence and categorises it. Its implementation simply measures the Lines Of Code (LOC) of each program state "P_i" where less lines mean a higher fitness. The algorithm evaluates each neighbouring transformation sequence and changes their order based on its fitness and position within the search-space [29]. Based on this assumption, the algorithm changes the transformation sequences until the algorithm cannot exchange the order anymore. After this step, the algorithm assumes the chosen transformation sequence is the best. This tactic can be utilised several times on the generated search-space, and travels from a local to a global optima.

2.5.3 Genetic Search

Genetic algorithm searches are population-based search tactics which start from an initial randomly chosen population and evolves over server generations, to the best solution. Individuals in successive generations have a better or at least no worse fitness values than those in preceding generations [30]. Genetic algorithms initiate the evolution with the aid of operations of **crossover**, **selection** and **mutation**:

- **Selection**, is the section process of choosing two individuals on the basis of their fitness. These two are used for the first crossover.
- **Crossover**, is the process of exchanging information between two individuals. This results usually in the division of each individual at a selected position. By swapping the adjacent sides across, a new set of individuals is created.
- **Mutation**, is the process of introduction diversity into the populations. This is achieved by the provision to let chromosomes alter. The probability that a chromosome will be altered is referred to as the mutation rate [30].

These evolutionary operators are applied iteratively across each new population, the genetic algorithm terminates after a finite number of generations. The approach used by [28] applies this technique on the evolution of transformation sequences.

To map this to the application of transformation sequences, a transformation can be considered as a chromosome, whereas a sequence of transformations is an individual.

Two transformation sequences are two individuals. The crossover between two individuals is chosen on a random basis and their mutation can be considered as a randomly chosen transformation within a transformation sequence. The result is a transformation sequence which is usually not applicable. Compared to the hill-climbing algorithm the genetic tactic achieves far better program transformation results.

2.5.4 Prediction based Program Transformation

The previous section presented two program transformation search tactics to find applicable transformation sequences. However with the generation of a huge search-space these approaches are inefficient. Shayun Li proposed a prediction based approach which utilises software metrics to model program transformation targets. A complex prediction technique calculates applicable transformation sequences which should lead to the overall reengineering target. The approach is based on a transformation process model to calculate applicable transformation sequence. The generated transformation sequence search-space is evaluated based on a defined reengineering goal. The model represents transformation sequences as tree structures, in which at least one of the leafs transformations is applied within a transformation process. Similar transformation sequences are combined and represented within a tree node as an alternative. The tree is traversed from the root node towards its children, until a leaf has no outgoing edges. At each tree node, the specified target is evaluated. Based on these targets, the tree is traversed in a particular direction and at each tree node, a transformation is tried to be applied. None satisfying program transformation results are ignored. At the end the final program state is evaluated based on the defined overall target. An example could be the evaluation of the LOC of a program source. The presented technique leads to better transformation process results than the presented search based tactics. However the approach is lacking, as it does not take all specified transformations into account. Because of this, specified reengineering targets cannot be guaranteed to be fulfilled and interplay effects, which are utilised to prepare a program source with program transformations for a particular program specification, are totally ignored [31].

2.6 Parallel Computing

In the 1980s it was believed that computer performance could best be improved by creating faster and more efficient processors. This theorem was augmented by the development of parallel processing techniques, to combine the computing power of two or more computers to jointly solve computational problems. Since the early 1990s there has

been an increasing trend to move away from expensive and specialised parallel supercomputers, vector-supercomputers or massively parallel processor types, towards cluster architectures based on common PCs, Workstations or SMPs. One of the major driving forces which have enabled this transition is the rapid improvement in the availability of commodity high-performance components, PCs and Workstations [4]. In the past, parallel programming and the exploitation of architectural properties of parallel machines was limited to a narrow field of experts. Because of the hardware industrys shift towards parallel computing, today's programmer and algorithm developer are challenged to utilise and understand parallel computer architectures and their theory. The following sections present an overview of their important attributes.

2.6.1 Parallel Application Development

The class of applications parallel processing environments can typically cope with fall into the categorisation of the demand of sequential- and grand challenge/supercomputing-applications. Grand Challenge Applications (GCAs) try to solve fundamental problems in science and engineering with a broad economic and scientific impact [32]. They are considered to be unsolvable without parallel computers. A typical example of a grand challenge problem could be the estimation of the global climate change or the simulation of some phenomena that cannot be evaluated through experiments.

The development of parallel applications is a complex task. They largely depend on the availability of adequate parallel software tools and environments. Software programmers must understand the problem domain which also includes the transformation of sequential to parallel applications. During parallel application development, many circumstances such as communication, synchronisation, data partitioning, distribution, load-balancing, fault tolerance, as well as deadlocks and race conditions have to be resolved. These issues need to be considered and evaluated to establish efficient parallel computation. Today, the number of specialised programmers who have knowledge of parallel and distributed computing techniques is still limited. Parallel computation can only be successfully accomplished by solving the following expectations [33]:

- Provision of architecture and processor transparency.
- Provision of network and communication transparency.
- Easy to use and reliable parallel programming techniques.
- Provision of fault tolerance.
- Assure portability.

- Support for traditional high-level languages.
- Increase of performance.
- Parallelism transparency.

Some of the presented problem domains are already solved. In general, internal details and the behaviour of the underlying architecture should be hidden from the user perspective, and programming environments should support high-level parallelism. Otherwise, if provided parallel programming interfaces are too complicated to use, parallel applications will be highly unproductive and cumbrous to use. Basically there are two existing parallel programming philosophies:

- **Implicit parallelism** is followed by parallel languages and paralleling compilers. The user can not specify and thus cannot control the scheduling of calculations, nor the placement of processing data.
- **Explicit parallelism**, is the process in which the programmer is in control of most of the parallelisation process and its communication structure. This technique is based on the assumption that the user is often the best judge of how parallelism can be exploited within the parallel application domain.

2.6.2 Code Granularity and Levels of Parallelism

In modern computing, parallelism can be achieved at various levels in both hardware and software components. For example parallel signal travel can be considered as the lowest form of parallelism whereas at a slightly higher level, within instruction level, multiple functional components could operate in parallel to achieve better performance. A sample could be a PC processor which processes three instructions in parallel. At a slightly higher-level, Symmetric-Multi-Processing (SMP) systems use multiple CPUs to work in parallel. At an even higher level, several computers can be connected to a single machine to perform parallel tasks, commonly known as cluster computing. The first two levels of parallelism, signal and circuit, are performed by a hardware implicit technique, known as hardware parallelism. However, within the last two levels, component and system behaviour are mostly expressed *implicitly* or *explicitly* by various software techniques. This process is formally known as software parallelism. Levels of parallelism can also be categorised by "chunks of source code". The separation and granularity of code has one common goal, to boost processor or parallel computing efficiently. The main drive for this behaviour is to execute two or more single-threaded applications in parallel. Common computer applications would be compiling, text formatting, database searching, device

simulation or parallel applications with multiple tasks. Parallelism in applications can be detected at several levels [34]:

- Very-fine grain, multiple instructions.
- Fine-grain, data level.
- Medium-grain, control level
- Large grain, task level.

The first two levels of parallelism are usually handled by either hardware or parallelising compilers whereas the last two are supported by programmers. The most important models utilised for developing parallel applications are shared-, distributed- (message-passing) and distributed-shared memory models.

2.6.3 Parallel Programming Models and Tools

Different parallelising models exist, this section reflects the most common ones [33]:

- **Parallelising Compilers** try to explore regular parallelism within software programs. They attempt to efficiently parallelise loops. Their use within applications on shared-memory multiprocessors and vector-processors has been quite successful. However they face difficulties within parallelisation of distributed environments, because of their non-uniform memory access time.
- **Parallel Languages** did not have the breakthrough for parallel program application programmers. They are usually not willing to learn new parallel languages, they rather prefer their traditional high-level languages as C or Fortran and focus on reengineering their existing sequential software. Parallel computing library extensions for existing languages or run-time libraries are the only practicable alternatives.
- **Message Passing Libraries** allow efficient parallel programs to be written for distributed memory system. These libraries provide routines to initiate and configure the message-passing environment as well as sending and receiving packets of data. The currently most popular high-level message passing systems for scientific and engineering applications are PVM and MPI. These environments give the user a comfortable tool to write programs for every existing computing platform without the need to rewrite their applications. The goal of portability, architecture and network communication has been achieved with low-level libraries. However

the difficulty still exists within its low-level parallelisation process specification, which is still left to the application programmer. Leaving the challenge to write most of the communication infrastructure, synchronisation and data partition, can be a burden. Tools which could assist and specify theses tasks are rare.

- **Parallel Object-Oriented Programming** stands for the provision of suitable software engineering methods for structured parallel application design. Similar to the traditional object oriented model, objects are defined as abstract data types which encapsulate their internal states through well-defined interfaces. They are considered as data containers. With this definition model, the objects can be treated as a collection of shared objects within a parallel environment. In todays computer industry, object-oriented programming is state-of-the art whereas the move slowly goes towards parallel object-oriented programming environments. However the lack of its acceptance is still due to the fact that traditional program developers still like to write their programs in old fashion programming languages such as Fortran. However is has been considered to be a promising technique for future parallel computations.

2.6.4 Methodical Design of Parallel Algorithms

There is no simple recipe for designing parallel algorithms. The methodology approach offers the maximisation of options for a program developer with provision of alternatives by reducing the cost of backtracking wrong choices. The design methodology allows the programmer to focus on machine independent issues such as concurrency within the design process whereas machine specific aspects of design can be delayed until late within the design process [35]. This methodology organises the design process into four stages. The first two stages seek to provide scalable algorithms whereas the last two focus on locality and performance related concerns:

- **Partitioning** refers to the decomposing of the computational activities and data on which it operates into several small tasks. The decomposition which focuses on partitioning data with a problem domain is known as the domain/data decomposition whereas the computation into disjoined tasks is known as functional decomposition.

- **Communication** focuses on the aspects of the flow and coordination of information among the tasks that are created during the partition stage. The nature of the problem and the decomposition method determine the communication pattern

among these cooperative tasks of parallel program design. Four common communication pattern exist: local/global, structured/unstructured, static/dynamic and synchronous/asynchronous.

- **Agglomeration** is the stage in which the tasks and communication structure defined are evaluated in terms of performance requirements and implementation costs. If required, tasks are grouped into larger tasks to improve performance or to reduce development cost. These methods will help to reduce communication costs by increasing the computation and communication granularity, gaining flexibility in terms of scalability, mapping decisions while reducing software engineering costs.

- **Mapping** is focused on assigning tasks to a processor which is maximised on the utilisation of system resources while minimising the communication. Mapping decisions are either considered statically (at compile time) or dynamically at runtime by load-balancing methods.

2.6.5 Flynn‘ s Classification

There are many ways to classify parallel computers. The most common one is Flynns Taxonomy by Michael J. Flynn. It classifies architectures based on *instructions* and *data* of which both can only exist in one of the two states, *single* or *multiple* [36].

- **SISD:** Single Instruction Single Data is the classical von-Neumann mono-processor system architecture in which a single processor executes single instructions on data stored in a single memory.

- **SIMD:** Single Instruction Multiple Data is one way how data parallelism can be achieved. The first supercomputers of this kind were vector or array processors.

- **MIMD:** Multiple Instruction Multiple Data is another way how parallelism can be achieved. MIND machines consist of a number of processors that run asynchronously and independently. The involved processors can execute different instructions on different pieces of data. Therefore MIMD architectures are used within different application fields as computer-aided design / computer-aided manufacturing, simulation, modeling, or within communication switches utilising shared or distributed memory.

- **MISD:** Multiple Instruction Single Data, is another parallel computing paradigm in which many functional units perform different operations on the same data. There do not exist many implementations of this type of architecture, as MIMD

and SIMD are more often appropriate for common data parallel techniques but they are harder to realise due to the difficult pipeline concept they have to inherit.

2.6.6 Parallel Programming Paradigms

It has been widely known that parallel applications can be classified according to the defined programming paradigm. Paradigms are used to repeatedly develop many parallel programs. Each paradigm is a class of algorithm that have the same control structure [34]. The choice of paradigm is determined by the availably of parallel computing resources and the level of granularity which are efficiently supported by the system. The type of parallelism reflects either the structure of an application or the data or both types. The choice of typical distributed applications are based on the popular client/server paradigm. In this parallel computing environment processes usually communicate through RPCs. The following paradigms are popular within the field of parallel computing:

- The **Task-Farming (Master/Slave)** paradigm utilises two different entities: master and slave/s. The master computer node is in charge of decomposing the problem domain into tasks and distributing them between the work nodes. Gathering the partial results in order to produce the final result of computation belongs also to its domain. The slaves process simple tasks, receiving the message with the task, process the task and send the computation back to the master. The communication usually only takes place between the master and the slaves. Task farming uses either static- or dynamic load-balancing. In the first case, the distribution of tasks is performed at the beginning of the computation, in which the master node participates in the computation after all work has been distributed. The allocation of work is based on the load-balancing or scheduling algorithm. Another technique is to use the dynamic load-balancing paradigm, which is commonly used when the number of tasks exceeds the number of available processors, or the number of tasks is unknown at the beginning, the execution time is not predictable or unbalanced problems occur. The important feature of the system is the dynamic adaptation of the system to changing environment conditions, not only the load of the processors, but also the possibility of the reconfiguration of the system resources. Due to these characteristics the system can respond quite well to failure of some processes and therefore insures robust applications development.

- **Data Pipelining** can be considered as more fine-grained parallelism based on functional decomposition . The tasks of the concurrent application are identified and each processor executes a small problem of the global problem domain. The

pipeline algorithm is one of the simplest, common and most popular functional decomposition paradigm. Processes are organised within a pipeline and each processor operates within a particular stage on the assigned task. The communication pattern can be very simple, since the data follows within the adjacent directions of the pipeline, also known as data flow algorithm. The communication can be completely asynchronous. The efficiency of this construction mostly depends on the load-balancing ability across the stages of the pipeline.

- **Divide and Conquer** is derived from the sequential paradigm and refers to the division of a problem into sub-problems. Each of these sub-problems are solved independently and the results are combined to produce the final result. Processing may result in the division or combination of sub-problems. Within parallel divide and conquer the results can be computed in parallel to be more efficient.

- **Speculative Parallelism** is utilised when it is hard to evaluate parallelism within the application domain, through any of the above stated paradigms. This algorithm is facilitated where it is hard to discover parallelism due to dependencies within the sequential application field. The result is that the problem is chunked into smaller parts and computed concurrently, utilised by some speculation or some optimistic parallelism calculation.

2.6.7 Parallel Performance Evaluation

The build of parallel software versions enables applications to run a given data set in less time. The success of the parallelisation process can be typically quantified by measuring the speed-up of the parallel version relative to the serial version. In addition to that comparison, it could be also useful to compare the relative speed-up to the upper limit of the potential speed-up [37].

Performance prediction can be evaluated based on the work each processor can perform. Additionally this also includes the idle time for interprocess communication. In general, work relates to computation as well as to data transfers. Intercommunication is based on synchronisation, message-passing or queuing delay, which can be considered as performance degradation. Traditionally performance modeling techniques have been based either on deterministic complexity analysis or some form of queuing analysis. Complexity analysis considers the evaluation of the amount of work and the extension of the synchronisation of the parallel processes, whereas state-of-the-art queuing analysis is focused on the amount of work (generated search space) lined up for parallel processing.

The faster an application performs, the less time the user has to wait for results. Shorter execution time also allows users to compute large data sets in an acceptable amount of

time. The computed value that offers a comparison of serial and parallel execution time is speed-up. Speed-up can be considered as the ratio of serial execution time versus parallel execution time. To give an example, if the serial computation executes in 3400 seconds and a corresponding parallel application runs in 60 seconds using 60 concurrently working processes, the speed-up of the parallel application is 57X (3400/60 = 56.666). For a given application domain that scales well, the speed-up should increase at, or close to, the same rate as the increase in the number of processes involved. When increasing the number of processes used to scale the application and if measured speed-ups fail to keep up, level out, or begin to go down, the application doesnt scale well on the data sets measured. Speed-up can be considered as a metric to measure efficiency. While speed-up is a metric to determine how much faster parallel is versus sequential computing, efficiency indicates how well software utilises the computational resources of the parallel system. The parallel execution efficiency can be simply calculated by dividing the observed speed-up by the number of processes utilised. The result is the efficiency in percentage. For example, a 57X speed-up on 20 cores equates to an efficiency of 95% (57/60 = 0.95). This means that, on average, over the course of the execution, each of the processors was idle 5% of the time. That issue can be addressed using Amdahls Law and Gustafsons Law.

2.6.7.1 Amdahl‘s Law

The Amdahl Law [38] mathematically specifies the speed-up that can be expected from parallelising serially performed tasks, on a parallel architecture. Amdahls law, named after computer architect Gene Amdahl, is used to find the maximum expected improvement of a system when only parts of the system are improved. It is often used in parallel computing to predict the theoretical maximum speed-up utilising multiple processors. The generalisation of Amdahls law is [39]:

$$Speedup(N) = \frac{1}{(1-p)+(\frac{p}{N})}$$

with its constants:

- **p:** Percentage of an algorithm that can be parallised.
- **N:** Number of processors involved.

Amdahls law is a formula that computes the expected speed-up of a parallelised implementation relative to a non-parallelised algorithm implementation. For example: An

algorithm from which 14 % of the involved operations could be parallelised, therefore run faster, while the rest of the operations (86 %) still compute at the same speed, are not parallelisable, the maximum speed-up of the parallelised version with 2 processors with Amdahls law calculation would be:

$$Speedup(N) = \frac{1}{(1-0.14)+(\frac{0.14}{2})} = 1.0.75X$$

The speed-up calculation states that the program would run 7,5 % times faster. The law simply focuses on the maximum achievable speed-up through a parallelisation process.

2.6.7.2 Gustafson's Law

Gustafsons Law, states that problems with large, repetitive data sets can be efficiently parallelised. Gustafsons Law disagrees the Amdahls law, which describes a limit on the speed-up that parallelisation can provide. Gustafsons law was first described by John L. Gustafson and Edward H. Barsis and is specified as [40] [41]:

$S(P) = P - (\alpha \cdot (P - 1))$

where *"S"* stands for the speed-up calculation of *"P"* for the number of processors involved, minus the non-parallelisable part of the process. Gustafsons law addresses the insufficiency of Amdahls law, which does not scale the availability of computing power as the number of machines increases. Gustafsons Law proposes that programmers set the size of problems to use the available equipment to solve problems within a practical fixed time. If faster, more parallel equipment is available, larger problems can be solved in the same time. In comparison to this, Amdahls law is based on fixed workload or fixed problem size. It implies that the sequential part of a program does not change with respect to machine size, or number of processors.

The impact of Gustafsons law was to shift research goals to select or reformulate problems so that larger problems are solved in the same amount of time. In regard to this, the law redefines efficiency as a need to minimize the sequential amount of a program, even if an increase of the total amount of computation occurs. For example, if 1% of execution time on 32 processors is spent in serial execution, the speed-up of this application over the same data run on a single machine would be:

$S(P) = 32 - (0.01 \cdot (32 - 1)) = 32 - 0.31 = 31.69$

With the same data, the equation of Amdahls Law would state $1/(0.01 + (0.99/32)) = 24.43$. This is a false computation, since the given percentage of serial time is relative

to the 32 processes computation. Further assuming that the total execution time for a parallel application is 1040 seconds for 32 processors, then 1% of that time would only be serial, which would be equal to 10.4 seconds. By multiplying the number of seconds (1029.6) for parallel computation on 32 processes, the total amount of work done by the application takes 1029.6*32+10.4 = 32957.6 seconds. The sequential time would be 10.4 seconds, 0.032% of the total work time. Using these proper values, Amdahls Law would calculate an overall speed-up of:

$$Speedup(N) = \frac{1}{(0.00032)+(\frac{0.99968}{32})} = 31.686$$

Since the percentage of serial time within the parallel execution must be known for the use of Gustafsons Law, a typical usage for this formula is to compute the speed-up of the scaled parallel execution to the serial execution of the same sized problem. Demonstrated within the above examples, a strict use of the data of the application executions within the formula for Amdahls Law gives a much more pessimistic estimate than the scaled speed-up formula.

2.6.8 Scheduling

Scheduling can be considered as a process of deciding on how to allocate resources for possible tasks. A scheduling algorithm in computer science is a method which gives threads, processes or data flows access to system resources. This is usually supported by a load balancing system. The drive for scheduling algorithms arose from the requirement of modern computing systems to perform multiple tasks at the same time (multitasking). Scheduling algorithms can help to minimise resource starvation while they try to ensure the fairness among the participants. Scheduling is confronted with the problem of deciding which of the waiting service requests should be allocated to which system resource. There exist many different scheduling algorithms within this field. The main ones are [42]:

- **First In First Out:** The simplest scheduling technique is the First Come First Served (FCFS) algorithm. It simply queues and executes processes in the order they arrive in the system queue.
- **Shortest Remaining Time:** Shortest Job First (SJF) is a scheduling strategy, in which the system scheduler arranges processes with the least estimated processing time remaining to be next in the queue system. This usually needs advance knowledge or estimations about the time required for a process to complete its task.

- **Fixed Priority Pre-Emptive scheduling:** Commonly used within OS to schedule processes, and is focused to assign a fixed priority rank to every system process, and the scheduler arranges the processes in the queue in order of their priority ranking. This causes lower priority processes to get interrupted by incoming higher priority processes.
- **Round-Robin Scheduling:** The process scheduler assigns fixed time slots to queued process, and cycles through them.
- **Multilevel Queue Scheduling:** This technique is used for situations in which processes can be divided into different groups. Common distinctions are made between foreground (active) and background (batch) processes. The reason for this type of scheduling is, processes can have different response-time requirements and so may have different scheduling needs.
- **Exclusive:** This scheduling technique minimises the queuing circle flow, by assigning only one process at a time to each system resource.
- **Fairshare:** This scheduling technique is based on knowledge and considers strategies, protocols and processing time, scheduled processes (tasks) may consume. The result is that, processes which have not used the system resource so commonly have a higher priority within the queue to access the system resources than recent system processes.

2.7 Parallel and Distributed Computing

The terms parallel computing and distributed computing do not have a clear distinction because they overlap. The same system architecture may be characterised as both parallel and distributed, processors in typical distributed system run concurrently in parallel. Parallel computing could also be considered as a tightly-coupled form of distributed computing whereas distributed computing may be considered as a loosely-coupled form of distributed computing. As a general classification the following criteria have been specified [43]:

- In a parallel computing architecture, all processors have access to the same shared memory. Shared memory can be used to exchange information between the processors.
- In a distributed computing environment, each processor has its own private distributed memory. Information is exchanged by passing messages between the processors.

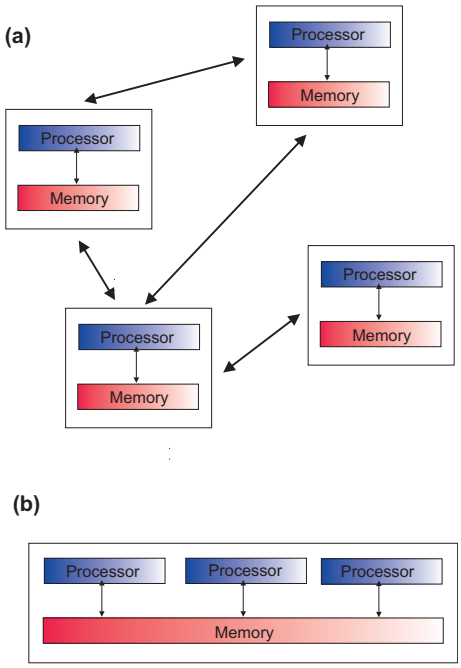


FIGURE 2.1: Distributed vs. Parallel Computing

The Figure 2.1 illustrates the differences between (a) distributed computing and (b) parallel systems. Within a distributed computing environment the system usually communicates via a message passing system whereas in (b) each processor has a direct access to the shared memory.

2.7.1 Parallel Application Domain

There are two main reasons for utilising parallel computing or distributed computing techniques. The first lays in the nature of application which may require the use of a communication network that connects several computers. As an example, data is produced in one machine and it is needed in another location. The second reason is that there are many cases in which the use of a distributed or parallel system would be more beneficial to the application domain. For example, a cluster system of several low-end computers would be more cost-efficient in comparison to a single high-end computer. On the other hand, a distributed computing system could be more reliable, easier to expand and manage than a single machine operating system, because there is no single point of failure. Examples of distributed and parallel computing system domains are:

- **Telecommunication Networks:** Telephone networks, computer networks such as the Internet.

- **Network Applications:** Distributed databases, network file system, peer-to-peer networks.
- **Real-Time Process Control:** Aircraft control or industrial systems.
- **Parallel Computation:** Scientific computing, weather forecast, cluster computing or grid computing.

2.7.2 Clustering Systems

Computer clusters can be considered as a group of coupled computers which work closely together so that in many respects they can be viewed as a single system. Its components are commonly connected through fast local area networks. Clusters are usually deployed to improve performance and/or availability over a single computer. They are typically much more cost-effective in comparison of speed or availability [44]. Cluster systems are traditionally used for technical applications such as simulations, biotechnology, financial market modeling, data mining, stream processing, serving audio, or games through the Internet. The basis for the cluster computing theory was originally laid by Gene Amdahl of IBM who published a paper on parallel processing called Amdahls Law [38].

2.7.3 Clustering Categorisations

- **High-Availability (HA) Clusters:** HA clusters serve the purpose to improve the availability of services within a cluster system. This includes the accessibility of redundant compute nodes when system components fail. The most common cluster size is two compute nodes, which is also the minimum requirement to provide redundancy. There are many commercial implementations of HA clusters available. A free and open-source version is Linux-HA for the Linux operating system.
- **Load-Balancing Clusters:** The aim of a load-balancing cluster is to evaluate the optimum performance gain for any assigned compute work. To achieve this, redundant node services are provided, analysed and utilised for the computation. The clusters load-balancer evaluates the optimum performance of each compute node and assigns the work to the best available resource. Commercial software solutions are the Sun Grid Engine, Moab Cluster and Maui Cluster Scheduler.
- **High Performance Cluster (HPC) Clusters:** HPC clusters are designed for performance while computational tasks are split across its compute nodes. Most scientific programs are designed for parallelisation and optimisation of cluster

workload. The most common HPC implementation is the Beowulf cluster, utilising the free available Linux OS and common Message-Passing-Interface (MPI) libraries for communication purposes.

- **Grid Computing:** Grid computing can be considered as applying the resources of many computers in a network to a single problem. Scientific or technical problems are solved, requiring a great amount of computer processing power or access to large amounts of data. Grid computing requires the use of software that can divide and farm out pieces of a program to as many as several hundred computers. It can be considered as a form of network-distributed parallel processing and large-scale cluster computing. The most common example of grid computing within the public domain is the Search for Extraterrestrial Intelligence Search for Extraterrestrial Intelligence (SETI) @Home.

2.8 Cluster Management Software

To fully utilise the capabilities and performance of a cluster, a cluster management software system is indispensable. Cluster management software can be considered as a back-end Graphical User Interface (GUI) or a command-line tool that runs either on the headnode or on a separate cluster node, the management server. Cluster managing services usually communicate and collaborate with cluster node agents. They run within each computing node and manage and configure local node services. Cluster managers are used to dispatch work to the cluster nodes. If in other cases the cluster is more related to availability or load-balancing than to computational demand, special clusters services are needed.

2.8.1 Linux High-Availability (HA) Cluster Manager

As Linux is growing to handle large server systems it has to provide many features which are supported by large server system providers such as Sun, Compaq, IBM and many others. One of the key features these large systems have in common is HA clustering. An HA cluster can be considered as a group of computers which collaborate in such a manner that the failure of a single cluster node will not cause the unavailability of the services its cluster provides. Given this definition, it seems obvious that it is necessary for the cluster management system to detect when servers or some of its services fail and when they become available again. In the case of an HA cluster manager, this function is executed by a program known as the heartbeat of a cluster node. Heartbeat programs typically send packets via the TCP/IP protocol to each machine to determine if they are still

available. If the heartbeat of a cluster node stops, databases or node service states are automatically saved and mapped to a redundant node for further processing. The Linux HA cluster manager [45] is one of this kind, automatically managing computer clusters and its services. Initialising the cluster, monitoring its health, recognising node failure or regulating the reformation of the cluster when a node joins or leaves the system are its advances. Each cluster node is controlled by a small daemon process which monitors its startup process and also serves as a central point for inter-node communication.

2.8.2 Oracle Cluster Manager for Solaris

Oracle cluster manager [46] utilises an agent based system to recognise and control node failure and to start new cluster services. The environment is equipped with a GUI to graphically display cluster information, monitor configuration changes and to check the status of its components. The system has the feasibility to perform many administrative tasks which are provided by Oracles Management Center. However this module is not capable of performing any specific task. This needs to be done via a command line. The suite relies on commonly used Simple Network Management Protocol (SNMP) to control and observe network components such as routers, switches, servers, printers, computers and workstations. To save all collected data, the management tool utilises the Management Information Base (MIB) standard to monitor and protocol all cluster services or devices within the system. With this protocol and description language, the management services are able to collect and understand environment data and are therefore able to evaluate it for further cluster analysis.

2.8.3 Veritas Global Cluster Manager

Veritas Global Cluster Manager (GCM) [47] is an add-on to the Vertias Cluster Server (VCS) package, and introduces HA capabilities to an integrated disaster recovery solution. The GCM provides the user with functionality of both a web-based management and a disaster recovery tool. The management tool allows the management of multiple Vertias Cluster Servers (VCSs) independent from their OS. The disaster recovery tool allows it to replicate and manage cluster services when a fail-over within the system occurs. The software provides access to the servers via a web-based GUI or Command Line Interface (CLI) tool, to monitor and administrate the clusters throughout an enterprise, whether they are local or geographically separated. The cluster manager can manage VCS clusters of all common operating systems of: IBM AIX, HP-UX, Linux, Oracle Solaris, and MS Windows.

2.9 Summary

This chapter discusses the related work that this thesis is based on. It provides an overview of software reengineering, describes its characteristics, explains program transformation and migration and shows examples of todays program transformation systems. It also reflects two parallel reengineering systems. It presents program transformation search- and prediction-tactics but also addresses the problems within this field. It reviews todays common parallel processing architectures, techniques and application fields. It further reflects the difference between parallel processing and distributed computing. Techniques to evaluate parallel speed-up, schedule cluster and monitor parallel systems are also described.

Chapter 3

Preliminaries

Objectives

- To present the FermaT transformation system and its transformation theory.
- To outline FermaT 's mathematical model.
- To give an overview and introduction of the Constraint Based Program Transformation Theory (CBPTT).
- To present and categorise constraints within a transformation process.

3.1 Introduction

One of the most challenging tasks a programmer can be confronted with, is the attempt to analyse and understand legacy assembler systems. Many features of assembler make its analysis difficult, and these are the same features which make migration from assembler to a high-level language a challenging task.

This chapter discusses the application of program transformation technology to assist the analysis and understanding of legacy assembler systems. The approach presented within this thesis utilises the FermaT transformation system for program analysis and program transformation application [18]. This section provides an overview of FermaT 's underpinning Wide Spectrum Language (WSL), its theory, its aspects and mathematical

foundation. Concept and techniques behind the Constraint Based Program Transformation Theory (CBPTT) [6] are outlined and illustrated by simple examples. The comprehension of both concepts is necessary to understand the parallel transformation processing techniques and ideas presented in this thesis.

3.2 FermaT Transformation System

The FermaT transformation system is a research-based industrial-strength formal transformation system which has been invented at Durham University [18]. The system can be used for program comprehension and language migration and is currently focused on transforming assembler code (IBM 360) to high-level maintainable C or COBOL program code [1]. The evolution of the FermaT transformation system started in 1989 through the invention of an academic prototype which only included a few number of program transformations. This tool was called Maintainer‘s Assistant (MA) and was implemented in the programming language LISP, and only aimed to test transformation ideas. It turned out that this tool was very successful in transforming assembler modules into equivalent high-level programs. Both transformations and resulting program result structures were represented in LISP.

Throughout the following years the whole system was restructured and reimplemented into its current language base of Wide Spectrum Language (WSL) and $\mathcal{META}$WSL. Both programming languages have been invented by Martin Ward at Oxford University [48]. They include abstract data types for representing programs as tree structures, as well as constructs for pattern matching, pattern filling and the possibility to iterate over its own components and program structures. The level of code abstraction can also be raised from a very low-level form, similar to assembler code, to a high-level representation which is comparable with languages like C or COBOL. Both languages cover the whole range of operations from general specifications to assignments, jumps and labels, which allows WSL, with this unique adaptability, to be an optimal language to apply code transformations during a restructuring process. The system also includes a $\mathcal{META}$WSL to Scheme translator to boot-strap the whole system to a Scheme implementation for further speed-up during a transformation process.

The latest version of this environment is called FermaT and embeds the FermaT transformation engine to apply program transformations [49]. Here, it has to be distinguished between a free-version known as fermat3, which can be downloaded directly from Martin Ward's homepage and a commercial version, declared as fermat2. The later version is also the core for the FermaT workbench [27]. FermaT's current transformation engine can be also considered as the foundation of the presented research and will be outlined

in further detail in the following sections. FermaT's transformation process to translate and transform legacy systems, consists of three basic steps [50]:

- Translation of legacy assembler code to WSL.
- Translate and restructure data declarations.
- Apply semantics-preserving WSL to WSL transformations.
  - For migration: Translate WSL to the target language.
  - For analysis: Apply slicing or abstraction operations to WSL to raise the abstraction level of the program source.

3.2.1 The Wide Spectrum Language (WSL)

Within the FermaT transformation system, program transformations are carried out in a wide-spectrum language while the transformations are written in the domain-specific programming language $\mathcal{METAWSL}$ [51]. These extensions include abstract data types for representing programs as tree structures as well as constructs for pattern matching, pattern filling and iterating over components of program structures. The transformation engine is implemented entirely in $\mathcal{METAWSL}$. The implementation of the $\mathcal{METAWSL}$ language also led to a $\mathcal{METAWSL}$ to Scheme interpreter for the realisation and presentation of abstract data types.

A program transformation in this context is an operation in which a WSL program changes its structural form, while its external behaviour stays equivalent under precisely defined denotational semantics. Due to FermaT's unique structural condition, program and specification consist within the same program language. Transformations can be utilised to demonstrate that a given program is correctly implemented according to its given specification.

Each WSL program is represented as an Abstract Syntax Tree (AST). By extending the WSL language with $\mathcal{METAWSL}$ constructs, its tree structures can be manipulated. This opens the possibility to express transformations in this specific $\mathcal{META}$ language. The result of this is, that FermaT's transformations can be used and applied on its own program source code, to test that it validates. Transformation results cause different program code structures compared to the original state, but its program behaviour stays the same. This results in the following advantages [52]:

- $\mathcal{METAWSL}$ is capable to maintain its own source code.

- $\mathcal{META}$transformations could be used on transformation code, in order to improve its efficiency.
- Prove theory on language oriented programming.
- The usage of transformations to restructure the transformation system.

3.2.2 The Kernel Language

The Wide Spectrum Language (WSL) has been grown from a simple and compliant kernel language to a complete and powerful programming language. The kernel language is based on infinitary first order logic [1]. Infinitary logic, originally developed by Carol Karp [53], is an extension of ordinary first order logic and allows conjunction and disjunction over countable infinite lists of formulae, and quantification over finite lists of variables. This opens the possibility to construct and express statements within the kernel language by combining infinitary logic formulae lists of variables and statement variables. Four primitive statements are needed, two of which contain formulae of infinitary first order logic, and three compound statements. So let "**P**" and "**Q**" be any formulae, and "**x**" and "**y**" be any non-empty sequences of variables. The following primitive statements are [50]:

1. **Assertion: "P"** is an assertion statement which acts as a partial "**skip**" statement. If the formula "**P**" is true then the statement terminates immediately without changing any variables, otherwise it aborts (abnormal termination and non-termination is treated equivalently, so a program which aborts is equivalent to one which never terminates).
2. **Guard: "[Q]"** is a guard statement. It always terminates, and enforces "**Q**" to be true at this point in the program without changing the values of any variables. It has the effect of restricting previous non-determinism to those cases which will cause "**Q**" to be true at this point. If this cannot be ensured then the set of possible final states is empty, and therefore all the final states will satisfy any desired condition, including "**Q**".
3. **Add variables: "add(x)"** adds the variables in "**x**" to the state space, if they are not already included, and assigns arbitrary values to them.
4. **Remove variables: "remove(y)"** removes the variables in "**y**" from the state space, if they are present.

These constructs express a rather pleasing duality between the assertion and guard statements, and the add and remove statements. The compound statements for the kernel language statements "**S1**" and "**S2**" would be specified as:

1. **Sequence: "(S1; S2)**" executes "**S1**" followed by "**S2**".
2. **Nondeterministic choice: "(S1 u S2)"** chooses "**S1**" or "**S2**" for execution, the choice is of a nondeterministic kind.
3. **Recursion: "(X:S1)"** where "**X**" is a statement variable (a symbol taken from a suitable set of symbols). The statement "**S1**" may contain occurrences of "**X**" as one or more of its component statements and represents recursive calls to the procedure whose body is "**S1**".

An example between WSL and the Kernel Language would be the following:

WSL Construct	**Wide Spectrum Language**	**Kernel Language**
Assignment	x:=1	add(<x>); [x = 1]
if-then-else	if B then S1 else S2 fi	([B];S1)([B];S2)

Table 3.1: WSL Example and Kernel Language

The listed **guard** statement may sound unfamiliar to many programmers whereas other programming constructs such as assignments and conditional statements seem to be missing at this point. The result of this implementation is that those missing atomic operations as assignments and conditionals can be constructed from the more fundamental constructs.

For example, the guard statement [**false**] is guaranteed to terminate in a state in which **false** is true. In the semantic model of FermaT, this is easy to achieve. The semantic function for [**false**] has an empty set of final states for each proper initial state. As a result, [**false**] is a valid refinement for any program. Morgan calls this a construct miracle [54]. These considerations have led to the development of the existing kernel language constructs, known as the "*Quarks of Programming*" [1]:

> "Mysterious entities which cannot be observed in isolation, but which combine to form what were previously thought of as the fundamental particles".

3.2.3 Semantics of a WSL Program

The mathematical model of WSL defines the semantics of a program as a function from states to a set of states. A state in this case simply represents a function which returns a

value from a given set "**H**" to each of the variables in a representing set "**V**" of variables. The set "**V**" in this case is the state space.

To be more precise, for each initial state "**s**" the function "**f**" returns the set of states "**f(s)**" which contain all possible final states of the program when it is started in state "**s**". The function "**f**" is called a state transformation, and represents a collection of symbols structured according to the syntactic rules of infinitary first order logic, and the definition of the WSL kernel language [55]. A special state "⊥" indicates non-termination or an error condition. A state predicate is a set of proper states other than "⊥" . If "⊥" is in a set of final states, then the program might not terminate for that initial state.

If two programs are both potentially non-terminating on a particular initial state, they are considered to be equivalent on that state. A program which might not terminate is no more useful than a program which never terminates. Therefore the semantic function is defined to be as such: whenever "⊥" is in the set of final states, then "**f(s)**" must include every other state. This restriction simplifies the definition of semantic equivalence and refinement. If two programs have the same semantic function then they are said to be equivalent [1, 50].

3.2.4 Weakest Precondition

Dijkstra introduced the concept of weakest precondition as a tool for reasoning about programs [56]:

> "For a given program "**P**" and condition "**R**", on the final state space of that program, the weakest precondition "WP(**P,R**)" would be the condition, which is the weakest condition on which the initial state "**P**" is started in. A state satisfying "WP(**P,R**)" is guaranteed to terminate in a state satisfying "**R**"."

Given any statement "**S**" and any formula "**R**" whose free variables are all in the final state space for "**S**" and all define a condition on the final states for "**S**". The weakest precondition "WP(**S,R**)" is defined to be the weakest condition on the initial states for "**S**", if "**S**" is started in any state which satisfies "WP(**S,R**)" then it is guaranteed to terminate in a state which satisfies "**R**". Utilising an infinitary logic reveals, "WP(**S,R**)" has a simple definition for all kernel language programs "**S**" and all (infinitary logic) formulae "**R**".

Kernel Construct	**Weakest Precondition**
Assertion	WP ({P},R) $=_{DF}$ P $\wedge$ R
Guard	WP ([Q],R) $=_{DF}$ Q $\Rightarrow$ R
Add Variables	WP (add(x),R) $=_{DF}$ $\forall$x.R
Remove Variables	WP (remove(x),R) $=_{DF}$ R

TABLE 3.2: Definition of "WP(**S,R**)" for the primitive statements of the kernel language

3.2.5 Specification Statement

For a reengineering system such as FermaT, it can be useful for both forward and reverse engineering processes to represent abstract specifications as part of the intermediate language. By the definition of specification statements, the refinement of a specification into an executable program, or the reverse process of abstracting a specification from executable code can be both carried out within a single language. Another advantage is that specification statements can be utilised in semantic slicing.

A specification describes what a program does without defining exactly how the program has to work. A specification could be expressed as a list of variables, in which the variables are allowed to change, and a formula defines the relationship between the old values of the variables, the new values, and any other required variables.

With this in mind, the notation "**x**:=**x'**.**Q**" is defined, "**x**" as a sequence of variables and "**x'**" the corresponding sequence of "primed variables", and "**Q**" is any formula. This assigns new values to the variables of "**x**" so that the formula "**Q**" is true (within "**Q**") and "**x**" represents the old values and "**x'**" represents the new values. If there are no new values for "**x**" which satisfy "**Q**" then the statement aborts. The formal definition would be stated in the kernel language as:

$$\mathbf{x}{:=}\mathbf{x}'.\mathbf{Q} =_{DF} \{\exists \mathbf{x}'.\mathbf{Q}\}; \text{add}(\mathbf{x}');[\mathbf{Q}]; \text{add}(\mathbf{x}); [\mathbf{x}{=}\mathbf{x}']; \text{remove}(\mathbf{x}')$$

It has to be kept in mind, that the specification statement is never **null**, so therefore the set of final states will be never empty for every initial state, when Dijkstra's: "*Law of Excluded Miracles*" is followed [57].

3.3 Transformation Scheme Descriptions for Transformation Processing

A reverse engineering process can be difficult and time consuming. During a reengineering process usually high-level information is extracted to be able to restructure

and reengineer source code to given software requirements. A few reverse engineering solutions have already been stated in Chapter 2.3.

The proposed approach utilises transformation schemes, constraints and FermaT's transformation system [49]. FermaT's mathematical proven program transformations are applied to restructure and simplify given Wide Spectrum Language (WSL) program sources. As mentioned in Section 3.2, FermaT uses its own intermediate language for restructuring purposes. A WSL program is internally represented as an Abstract Syntax Tree (AST). Referring to the context of program transformation application, FermaT's WSL code transformations are only applicable on specific AST types. After the fulfilment of applicability conditions, satisfying formal method rules, transformations are applied or rejected. It can be assumed, the bigger the WSL code rises, the bigger the search space for the application of program transformation grows. The following formulae computes the search-space "Σ" of possible transformations within a WSL program:

$$\Sigma = (t * p)^n$$

With "**t**" for the number of transformations, "**p**" for the number of AST nodes within the given WSL program and "**n**" for the transformation sequence length. A transformation sequence in this context defines a sequence of transformations. The order of transformations within this sequence defines in which sequence they should be applied.

For instance, if only 20 program transformations and 100 AST nodes within a WSL program are considered, and a transformation sequence length of 10 transformations has been selected, the resulting search space would be of: $20 * 100^{10}$ transformations. Within "*fermat3*", the open-source version of FermaT, 111 different WSL code transformations exist. By assuming the above calculation, the total search space would be: $111 * 100^{10}$ transformations.

To compute this enormous generated search space in reasonable time, parallel processing and transformation pre-processing techniques seem to be mandatory to lead to a successful reengineering solution in reasonable time. Nonetheless there still exists the problem on how to apply the transformations, with the following considerations:

- In which order should the transformations be executed?
- Which program transformation out of the FermaT transformation search space "Σ" should be used?
- How often should each transformation be applied?
- On which program state "P_i" should the transformations be considered?

- On which AST path of the given WSL program should the transformation be used?

Answering those questions seems to be only possible through the collection of sufficient information from the given WSL program. The process of applying program transformation is usually hard-coded and therefore not dynamic and adjustable enough. To describe and outline a transformation process in a dynamic way has led to the development of transformation scheme descriptions [6]. This completely new approach provides the maintainer with the possibility to completely outline a whole transformation process. Even further, a formal language has been developed to describe a transformation process. The transformation scheme language opens the possibility to narrow down the transformation search space. However the search space is still enormous, because of its generated transformation sequences. This approach is outlined in more detail in the following sections.

3.3.1 Transformation Schemes and Descriptions

As a standard does not exist within the applicability of program transformations [6], the transformation scheme approach has been designed for the purpose of bringing more structure to the applicability of program transformations during a reengineering process. A transformation scheme description can be described as a program transformation process outline with the following characteristics:

- A transformation scheme description consists of program transformations in a form of transformation sequences and alternative-constructs according to their Backus-Naur Form (BNF).
- Based on each WSL program transformation applicability condition, each program state "P_i" has to be saved.
- Within transformation scheme descriptions, constraints are utilised to guide and satisfy a defined reengineering goal.

To be able to assist within this transformation process a formal language has been developed. This language can be used by the software maintainer to describe and outline transformation processes. The purpose goes even a step further. The scheme description is also used to create an automaton, which generates the transformation sequence search space for transformation processing. A deeper understanding of this process can be followed within the thesis of [6]. The formal language used to describe transformation scheme descriptions is outlined within the next sections. The reason for this is,

one part of this thesis includes a transformation scheme description unrolling theorem, which proves and describes how transformation schemes description can be computed in parallel.

3.3.2 Transformation Scheme Description Language

As mentioned in the previous section a formal language has been developed to describe and outline transformation scheme descriptions [6]. The resulting automation which generates the transformation sequence search space from the description is not discussed within this approach.

The main focus has been laid on the development of transformation scheme descriptions. The developed language provides an opportunity to the software maintainer to write transformation processes in a simplified way. The difficulty lays in the understanding of transformation processes and the creation of the huge transformation sequence search space. The generation of transformation sequences is further discussed in Chapter 6. As illustrated with the small example at the beginning of this chapter, the transformation search space is generally huge. With the help of maintainer knowledge and the transformation scheme description language, this space can be narrowed. Even further, with the utilisation of parallel computing power which will be described in Chapter 7 this space can be even further decreased. The formal language to outline these descriptions consists of the following basic constructs as:

- A sequence construct which describes a process of lined up transformations as: T_i, T_{i+1}, T_2 .... T_{n-1}, T_n.
- Alternative transformation operations such as: $(T_i|T_j)$.
- A Quantifier and grouping construct: T_i [n..m].
- Constraints "C_n" or meta-constraints "*mC1*" to narrow down the transformation search space.

The transformation scheme language also known as Transformation Scheme Description Language (TSDL) [6] is presented below in its BNF:

BNF Type	**Definition**
<*scheme*>	<sequence>
<*sequence*>	<alternative> ("," <alternative>)*
	Continued on next page

BNF Type	**Definition**
<*computing nodes quantifier*>	“[” NATURAL NUMBER “]”
<*alternative*>	<factor> (“|” <factor>)*
<*factor*>	<subscheme> | <transformation> <quantifier>? <constraints>?
<*subscheme*>	“(” <scheme> “)”
<*transformation*>	“<” <meta_constraints> | FERMAT TRANSFORMATION (“@” AST PATH)? “>”
<*quantifier*>	“[” BINARY NUMBER “..” NATURAL NUMBER | # “]”
<*constraints*>	“{” CONSTRAINT (“,” CONSTRAINT)* “}”
<*meta_constraints*>	“{” (META CONSTRAINT (“,” META CONSTRAINT)*)? “}”
NATURAL NUMBER	a natural number (1 to n)
FERMAT TRANSFORMATION	a FermaT transformation
AST PATH	a path in the abstract syntax tree of the WSL code
BINARY NUMBER	a single-digit binary number (0 or 1)
CONSTRAINT	a constraint
META CONSTRAINT	a meta constraint
“,”	a comma to indicate a sequence

Continued on next page

BNF Type	**Definition**
"|"	a vertical bar to indicate a alternative
"("	a left bracket
")"	a right bracket
"<"	a left angle bracket
"@"	a separator between a FermaT transformation and an AST path
">"	a right angle bracket
"["	a left square bracket
".."	a separator between two numbers to indicate a mathematical interval
"#"	selected / determined number (1 to n)
"]"	a right square bracket
"{"	a left curly bracket
"}"	a right curly bracket

Table 3.3: TSDL Description in the Backus-Naur Form.

3.3.3 The Transformation Scheme Basic Constructs

This section introduces the usage of transformation scheme descriptions and its constructs. As explained, they serve the purpose to provide the maintainer with a tool to describe and outline transformation processes. To lead to a successful reengineering aim constraints are embedded. These specific constraints are explained in Section 3.4. Constraints are also encapsulated within transformation scheme descriptions to reduce the enormous transformation search space. Once a transformation scheme description is defined, the generation of the corresponding transformation sequences will be automatically performed. This process is described in Chapter 6. In regard to transformation scheme descriptions, the following entities can be found within their definition:

FermaT Transformations:
All transformations used within the expressiveness of transformation schemes and transformation scheme descriptions belong to the FermaT transformations catalogue. Their description and explanation can be found in Appendix A. How transformations and transformation sequences are applied within WSL specific programs is explained in Chapter 7.1.

Abstract Syntax Tree (AST) Path:
The AST path on which each FermaT program transformation can be applied can be treated as the core element of each transformation process. Without any path specified it is not possible to apply a transformation at all. To each transformation process belongs an AST path which indirectly tells the FermaT transformation engine in which WSL program section (AST node) the transformation should be applied. If it is the root-path of a WSL program code, symbolised as an "//", it has to be stated. However it has to be known that each transformation process can be expressed in a very dynamic manner, which is explained in Chapter 7.2.1.

Constraints:
Constraints are the most important factor in the whole transformation scheme description process. Constraints give the whole transformation process a satisfaction condition. These embedded constraints are checked at each WSL program state "P_i". The fulfilment of a stated reengineering aim can be regulated by satisfying or not-satisfying the specific program state. The process of satisfying a given constraint is described within the next section.

Meta-Constraints:
$\mathcal{META}$Constraint is another type of constraint which can occur within a transformation scheme description. An example of this expressiveness is demonstrated in Listing 3.1. The $\mathcal{META}$Constraint "*mC1*" could define the elimination of a "*Do-Loop*" within a WSL program. In this case, any FermaT transformation which eliminates this kind of loop is automatically chosen and applied. The search space would be equivalent to one FermaT transformation.

```
(
  < {mC1} @ // >
) {C1}
```

Listing 3.1: $\mathcal{META}$Constraint Transformation Scheme Description

Transformation Description:
Listing 3.2 shows an example of a simple transformation scheme description which contains one FermaT transformation. In this case, the transformation "*Simplify If*" would be tried to be applied on the AST path "//" (root- AST path) to try to satisfy constraint "*C1*". The constraint "*C1*" could for example include the achievement of less than 20 Lines Of Code (LOC) within the given WSL program code. The execution of this transformation scheme description would result in the appliance of the single transformation

"*Simplify If*" on the WSL program state "P_0", and results in a second WSL program "P_1". The difference between these two programs is their code structure. Of course the satisfaction of constraint "*C1*" can be only checked after the transformation process, by measuring the LOC.

```
(
  < Simplify If @ // >
) {C1}
```

Listing 3.2: Transformation Description

Transformation Sequence Description:

Listing 3.3 presents a transformation sequence description construct with two FermaT transformations of the same kind. In this case both transformations "*Merge Right*" like to be applied on the specified AST path "/0,1,2,3,6,1,0/", satisfying the fulfilment of constraint "*C1*".

```
(
  < Merge Right @ /0,1,2,3,6,1,0/ >,
  < Merge Right @ /0,1,2,3,6,1,0/ >
) {C1}
```

Listing 3.3: Transformation Sequence Description

Transformation Quantifier Description:

Listing 3.4 shows a transformation scheme description which states that the transformation "*Simplify Item*" should be applied between 1 and 10-times on the specified AST node type "*T_Assign*". By having defined this, the maintainer was probably not sure how many "*T_Assign*" constructs occur within the given WSL program code and how many transformations of this kind are necessary to fulfil constraint "*C1*". In this case the transformation scheme description will produce at least 10 different WSL program code versions on which constraint "*C1*" is evaluated. The evaluation process also depends on how many "*T_Assign*" AST types occur within each currently processed WSL program state "P_j".

```
(
  < Simplify Item @ T_Assign > [1 .. 10]
) {C1}
```

Listing 3.4: Transformation Factor Description

Transformation Alternative Description:
Listing 3.5 demonstrates an alternative-construct transformation scheme description with 4 FermaT transformations. Within the process of trying to apply this transformation scheme description, each cited transformation tries to satisfy constraint "*C1*". Further refined, the transformations "*Remove Recursion in Action*", "*Substitute and Delete*", "*Floop to While*" can be applied within the given WSL program on any AST node which fulfils the applicability condition. The only exclusion is the transformation "*Simplify Item*" which should be applied on the specified AST node "*T_A_S*", which stands for a Action-System within the WSL syntax.

```
(
  (
    < Remove Recursion in Action > |
    < Substitute and Delete > |
    < Simplify Item @ T_A_S > |
    < Floop to While >
  )
) {C1}
```

Listing 3.5: Transformation Alternative Description

3.4 Constraints in a Program Transformation Process

This section describes the definition, classification and use of constraints within the applicability of program transformations [6]. Constraints are utilised to guide through a transformation process. In particular they are used within the definition of a transformation scheme description discussed in Section 3.3. In order to lead to a successful program transformation application within the FermaT transformation system, constraints are categorised in their **behavior** and **characteristic**.

In the case of program transformation application, each program transformation changes the states and internal structure of the transformed WSL program. To ensure the success to a given reengineering target introduced by the maintainer constraints are introduced and categorised to underpin the reach of the specified goal. By introducing them within a transformation scheme description and eventually inside the generated transformation sequences, this leads to the *success* or *failure* of the whole transformation process. Consequently, the embedded constraints always try to make them reachable or not reachable by the maintainer defined goal.

To clearly specify those goals defined by the maintainer for a reengineering process, they have to be based and distinguished on properties (**Behaviour-Constraints**) and characteristics (**Structure-Constraints**).

The intention of this thesis is not the definition of constraints, they are just part of the parallel transformation process and are used to guide through the mostly automated process of applying FermaT program transformations in parallel. The developed constraints can be categorised into *Behaviour-* and *Structure constraints*. Each category consists of three sub-groups:

Low-Level-, **High-Level-** and **Environment-constraints**. Each group is more precisely defined by specific constraints. The result of this is that most of them are closely connected and they overlap. Figure 3.1 gives a graphical overview of the constraints definitions.

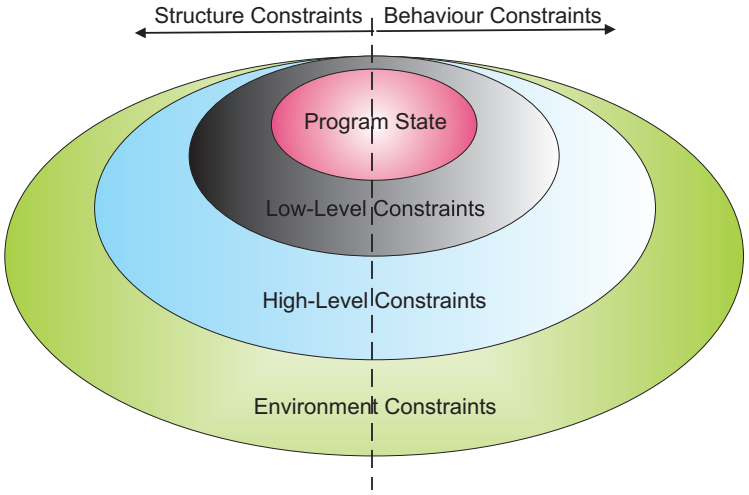


FIGURE 3.1: Definition of Constraints

3.4.1 Structure Constraints

Structure Constraints try to capture the static structure of a program and are defined as a characteristic or a set of characteristics of structural elements of a program. Structural elements of a program can be, for example, the maximum number of nesting depth as well as the Cyclomatic Complexity (CC) of a software program. For example a Statement within a given source can be categorised as a characteristic and is therefore a structural element of the Wide Spectrum Language (WSL). LOC and Number of Code Characters (NoCC) metrics can be classified as characteristics because their values have

an effect on the program source code or vice versa. Most of those constraints can be proved by an inspection. As an example: the Cyclomatic Complexity (CC) of 20 can be described as characteristics and are therefore a measurable constraint.

- **Low-Level Structure Constraints:** The introduction of Low-Level Structure Constraints specify or capture the atomic and local characteristics of a given WSL source code. As during a transformation process, the internal structure and therefore the Abstract Syntax Tree (AST) of the representing WSL program changes, and they can be easily captured. The definition of Low-Level Structure Constraints helps to monitor this satisfaction and to try to satisfy the estimated transformation goal. Examples of such are the Pattern and Convention constraints defined [6].

- **High-Level Structure Constraints** address global characteristics or a set of characteristics of the program source. In comparison to Low-Level and *Behaviour-Constraints*, other measuring techniques are needed to prove their correctness. The result of this is that High-Level constraints are often based on Low-Level constraints and depend on code structures such as data-types, control-flow, or the raise of abstraction level. High-Level constraints are defined as Metric, Abstraction-Level, Analysis or Data constraints.

- **Environment Structure Constraints** ensure that the structure characteristics of a program which are predefined by the environment are preserved. Compiler constraints or programming language constraints have been developed and are introduced within a transformation process, to avoid or limit the internal restructuring of the source program.

3.4.2 Behaviour Constraints

Behaviour Constraints are introduced in regard to the dynamic behaviour of the program to be reengineered. They are described as an individual property or a set of properties and can be classified as behavioural elements of a given program. Examples of such are execution time or memory consumption during runtime within a certain environment. They are proven to be correct when the property or properties of a program comprises a set or element of the given constraints. The complexity of a program can be categorised as a property which is measured on the basis of a metric. A constraint measuring the actual complexity of the given program state can be seen as a behaviour able one. Consequently special techniques are used to measure those dynamic properties.

- **Low-Level Behaviour Constraints** can be execution speed or memory consumption and are therefore properties of the behaviour of a program. As mentioned those properties have to be fulfilled to be a behaviour of the given program. The behaviour is based in on the internal structure of the program and is therefore very closely related to the given structure of the given WSL program.
- **High-Level Behaviour Constraints** address global characteristics or a set of characteristics of a program source. To their satisfaction, they either have to fulfil a global property or a subset of properties. These high-level constraints are based on low-level-constraints and can be proven as correct if the behaviour and the low-level constraints of the given program are satisfied. However these values are difficult to capture and are assumption-based. Examples of such would be metric or runtime constraints.
- **Environment Behaviour Constraints** are used to guarantee the adaptation of a program to a specific hardware or software environment. As FermaT and its proven program transformations are used to fulfil the aim to preserve program properties, the same properties are used to ensure the internal behaviour of the given program. Similar to the estimated high-level behaviour of a program source, this process is also assumption-based. Within a reengineering process, hardware and software constraints are often used in combination and ensure the satisfaction of the overall aim of the reengineering task.

A characteristic always regards the structure of a program whereas a property always regards its behaviour. The structure of a program has certainly a significant influence on the general behaviour of a program and vice versa. It is important to note that sometimes the only difference between a characteristic and a property is the context in which they are considered. Therefore it has to be noted that there are constraints which seem to belong to both classes, especially within the high-level and environment constraint classification. To give a simple example, a condition is a characteristic of a program because it is a structural element. It consists of characters and increases the complexity of a program [58]. The same condition can be transferred to a particular behaviour during its execution, which results in the final state the program terminates. So the condition can be also considered as a behaviour able element and therefore as a property. The same situation is applicable with a structure constraint to a time-critical loop which can have an impact on several different behaviour constraints. It has to be noted that the FermaT transformation system is only able to transform written WSL program source and therefore is not able to capture the behaviour of executable code during runtime. Environment constraints are introduced to clarify this situation. However, in most cases, it is obvious whether a particular constraint belongs to the

class of structure or behaviour constraints which makes the classification of them very traceable. More detail about the definition of constraints and the constraints based transformation processing can be found in the thesis of Natelberg [6].

3.5 Summary

This chapter has summarised the theoretical foundation on which this research is based. It reviewed the FermaT transformation system , its theory and its Wide Spectrum Language (WSL) [49]. It gave an overview and introduction of the Constraint Based Program Transformation Theory (CBPTT) and its transformation scheme descriptions. Transformation schemes descriptions can be utilised to define and outline a program transformation process. Transformation processes can become very complex and are not always understandable within the maintainer point of view. Utilising this technique more efficient transformation processes can be performed with less maintainer knowledge. Predefined and embedded constraints assist to fulfil a specified reengineering aim. Transformation scheme descriptions usually generate an enormous transformation sequence search space. These generated search spaces consist of thousands of transformation sequences which have to be tested and be applied.

Chapter 4

Parallel Transformations Framework: An Overview

Objectives

- To present the features of the proposed parallel transformations framework.
- To describe the parallel transformations system basic components.
- To present a formal language to specify parallel transformations tasks.
- To present the utilised parallel processing techniques.

4.1 Introduction

This chapter presents the initial concept of the parallel transformations framework proposed in this thesis. It outlines its main features, describes its architecture, its analysing and parallel processing system. A formal language to outline and specify parallel transformation tasks and parallel transformations processing behaviour is presented. Utilised software and hardware components are highlighted and parallel transformations processing techniques are introduced.

4.2 Parallel Transformations Framweork

Cluster systems have lately emerged as one of the major growth areas in applied computer science [3]. The past several years have witnessed an ever increasing acceptance and adoption of parallel processing, both for high performance scientific computing and for general-purpose applications. This trend was caused by the demand for higher performance, lower cost and sustained productivity, which lead to the acceptance of two major developments in parallel computing: massively parallel processors and the widespread use of workstations for parallel computing [44].

Todays programming languages, Java and C++ are becoming more and more popular [59]. However there still exists a considerable amount of code written in Fortran and COBOL [60]. The latter has been used quite frequently for programming business applications in banks, insurance companies and administration. Most of these applications were programmed for a centralised environment with mainframe computer and terminals. Todays companys face the challenge to take advantage of the more modern computing equipment of Personal Computers (PCs), laptops, and distributed and parallel environments. The result of this is that software systems have to evolve.

Parallel computing techniques can be considered as one way to fulfil this increasing computational demand of contemporary application programs. However, for a sequential application to benefit from parallel processing platforms, it has to be parallelised to take full advantage of multiple Central Processing Units (CPUs) [20]. If this work is done carelessly this may result in disadvantages, in the worst case scenario there could be a performance decrease or it may come to a halt [38]. Some of todays reengineering tools already have parallelism implemented, though it has to be distinguished between tools capable of [24]:

- Transforming sequential source code and parallelising it [20, 23] or
- Parallelising the transformation process [21].

This thesis focuses on the definition, analysis and parallelisation of transformation tasks. To fulfil these aims, the proposed framework utilises and presents the following features:

- Utilisation of cluster computing components for parallelising transformation tasks and processes.
- Development of an analysing system focused on parallelisation of transformations tasks with subsequent attributes of:

  - Parallel computing environment analysis.
  - Parallel transformation task analysis.
  - Transformation scheme description analysis.
  - Wide Spectrum Language (WSL) program analysis.
  - Transformation scheme description decomposition.

- Parallel transformation task computation and evaluation.

4.3 Architecture for a Parallel Transformations Framework

This thesis proposes a parallel transformations framework. At the start of this research project, parallel transformations processing was not possible within the FermaT transformation environment [5]. After an analysis of todays parallel computing environments, it seemed that the most appropriate solution would be to consider the classical Beowulf cluster style architecture [61] as the basis for the proposed approach. For the following reasons:

- An easy and flexible parallel computing environment.
- Utilisation and reuse of open source software packages and libraries.
- Computing resources can be added and removed during runtime.
- Cost efficient by utilising Commercial Off-The-Shelf (COTS) PC components.

How the proposed parallel transformations processing Beowulf style cluster architecture can be set-up is outlined in Chapter 8 (Tool Support). A more detailed version can be found within a separate available FCE tutorial. As dynamic is one of the underpinning attributes of this approach, the following basic principles have been taken into account:

1. Computing nodes can be added in an ad-hoc manner, during runtime to the environment.
2. Parallel transformation tasks are solved in an automatic manner.

An impression of how the presented approach on the basis of the utilised architecture functions is illustrated in Figure 4.1. The following briefly describes the parallel transformations processing basic work flow in its simplest form. How the developed parallel

transformations processing techniques operate can be followed in Chapter 7 and in more detail in Chapter 9, with concrete Case Study examples. As with every Beowulf style computer-cluster, the proposed parallel transformations system consists of the following architectural components: a **headnode** and **computing nodes**.

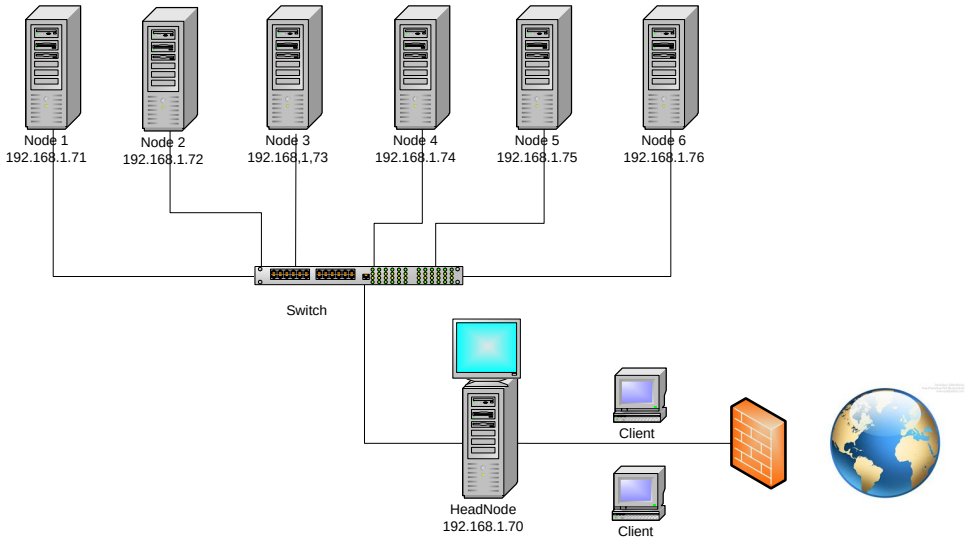


FIGURE 4.1: Parallel Transformations System

- The **headnode** which can be found at the bottom in Figure 4.1 can be considered as the master-node of the parallel processing environment. This node is in charge of all parallel transformation processes and delegates the other computing nodes. This element can be also considered as the core element of the developed system. Depending on the parallel processing technique specified by the maintainer with each task, the environment behaves differently. The system can distinguish between different job-submission processing modes. Firstly, a fully-automatic mode, in which precisely defined parallel transformation tasks are submitted to the headnode or as a second option, the manual mode, in which a maintainer can take influence on the parallel processing behaviour of the architecture. Both techniques are outlined in detail in Chapter 8 Tool Support. As mentioned, the main focus has been laid on the development of the headnode's knowledge. The headnode has to analyse the processing environment and each parallel transformation task, before a parallel transformation process is activated for computation. How parallel transformation tasks are defined by utilising the developed parallel transformation processing language, and how they influence the

parallel processing behaviour of the developed environment is demonstrated in Section 4.6. After the maintainer's submission of a transformation task specification, the headnode performs the following steps:

1. Analysis of the existing parallel environment.
2. Analysis of the specified transformation scheme description.
3. Analysis of the WSL program source to be transformed.
4. Decomposition of the transformation scheme description.
5. Submission of independent sub-tasks to the parallel environment.

The above techniques are comparable to similar pre-processing techniques used in other approaches such as data mining or machine-learning. How the analysis algorithm evaluates its information is outlined and described in Chapter 5.2. The satisfaction of predefined parallel processing constraints to speed-up the transformation tasks is also highlighted in these chapters. At the end of a parallel computation, the headnode usually collects all computing node computations, analyses & combines them and presents the result to the maintainer. Different task specifications could also state that the computing nodes should perform these steps. How this evaluation process performs can be followed in Chapter 7.

- The **computing nodes** can be found at the top in Figure 4.1 represented as worknodes. Depending on the evaluation and partition of each transformation task based on their embedded parallel processing constraints, each sub-task defined by the headnode can be computed differently by the computing nodes. The basis for this procedure is task supporting parallel processing constraints. This process is further described in Section 4.7.

On the basis of the evaluated transformation processing speed of each particular computing node, the headnode always tries to calculate the optimal workload-balance. Chapter 7 explains in more details how the computing nodes are connected for parallel transformations processing and how results are computed.

4.4 Awareness of Tasks Parallelism

The main intention of the presented parallel transformations framework is to provide suitable parallel transformations processing solutions in regard to any specified parallel transformation task outlined with the proposed parallel transformations processing language (Section 4.6). In most cases, parallel processing constraints embedded within the

parallel transformation schema are speed-up related. As cited in many papers, speed-up can only be achieved when parallelism is found. To detect parallelism, techniques for the decomposition and parallelisation of transformation scheme descriptions are presented and described. Their specifications can be found in Chapter 6. An introduction on how generated transformation sequences produced by the decomposition of transformation scheme descriptions are grouped and parallelised for parallel computation is also specified in the same chapter.

4.5 FermaT and a Parallel Transformations Framework

The extension of FermaT's theoretical model, for parallelising transformations had to be thought about very wisely. In consideration of the presented parallel transformations framework one of the most important questions was:

> "Would the parallelisation of transformation processes have any effect on FermaT's program transformations and its internal behaviour in preserving semantics?"

As stated in Chapter 3.2, if a program transformation is applied, the WSL program code structure changes while its semantics are preserved. The basis for such a behaviour is FermaT's mathematical foundation. Applying code transformations in parallel by chunking up WSL program code and preserving its semantic equivalence can be a challenging task. Program transformations ensure the semantic equivalence within their applicability domain and would also do this within each code chunk. But how can it be proved that parallel restructured code-chunks are equivalent to the original code? One solution leads to the evaluation of source code independence. However due to the fact that many hundreds of transformation sequences have to be applied on different WSL code versions, different AST paths or in this case separate WSL code chunks, the proof of source code independence would interfere and delay the parallelisation of transformations. Not even considering the network communication overhead generated by such a solution.

To remedy this situation the parallelisation is achieved from a different point of view. Each parallel transformation process is divided in such a manner, that the generated transformation sequences, are grouped and computed in parallel and results are evaluated based on the same WSL program start state "P_0". This reduces the communication overhead and opens the opportunity for more flexible parallel transformations processing solutions. The pipe-line-transformation process outlined in Chapter 7 is one solution.

This would also remedy the situation of implementing FermaT's internal transformation processing techniques in parallel. Within the proposed parallel transformations processing architecture, each computing node is equipped with a FermaT transformation engine instance. This opens up the possibility of being able to directly send commands to the computing nodes while at the same time, they have access to a network storage space which comprises all WSL sources and parallel transformation task files.

4.6 Parallel Transformation Task Description Language (PTTDL)

Within the domain of scientific parallel computing, various techniques for image-processing or signal processing have demonstrated that the definition of tasks for efficient parallelisation is indispensable. Once parallel tasks are identified and evaluated in relation to the processing aim, they can be mapped to a parallel processing architecture. With regard to parallel transformations processing, these techniques can be reused to map transformation tasks to a parallel environment.

To ensure this, task independence needs to be guaranteed. Each transformation scheme or sub-schemes decomposed and generated by the presented decomposition laws outlined in Chapter 6, can be considered as independent and hence computed in parallel. To assign dynamically or manually transformation tasks to the proposed parallel transformations processing environment, a formal task description language has been specified.

The language is designed to describe and outline primitive processes of parallel computation. By utilising the language, the maintainer is given a utility to describe concurrent transformations processes. The presented language enables the user to write and specify a collection of parallel transformation tasks, whereas tasks are executed concurrently based on system knowledge. The communication and the specification between processes are encapsulated within the developed metalanguage. Each process within an application specification describes a specific behaviour of a particular aspect of implementation, and communication channel describes connections between processes. The user can totally focus on the definition of parallel transformation task specification whereas the system evaluates a suitable parallel processing solution based on knowledge and specified rules. This approach has two important objectives. First, it gives the language specification a clear defined and simple structure. Secondly, it allows the parallel transformations processes application to exploit the performance of the system. However to assist and provide the system with knowledge, this language provides two important transformation task description constructs: *PAR* and *PLACED PAR*. The difference between these

two constructs is, the *PAR* construct defines a transformation task which will be analysed and internally lined up (inside the headnode) for parallel computation, whereas the *PLACED PAR* construct directly assigns a transformation task to a specified computing node. Their internal evaluation and processing workflow is demonstrated in Figure 4.2.

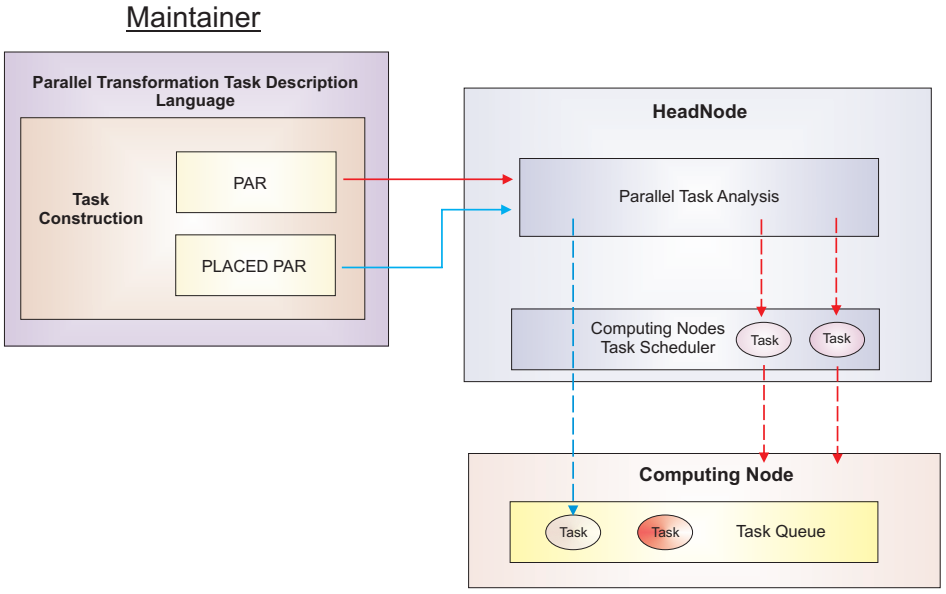


FIGURE 4.2: Parallel Transformation Task Description (PTTD) Workflow

Within both definitions, parallel transformation tasks have to be submitted to the parallel system's headnode. The headnodes analysing system evaluates each transformation task and either sends it directly to the computing node or further decomposes it based on its specification and specified decomposition laws. After the decomposition process, the generated sub-tasks are assigned and mapped to the parallel transformations processing environment. The computing node internal queuing system serves as buffers and regulates its transformations processing.

4.6.1 Parallel Transformation Task Definition

In terms of parallel transformations processing, a Transformation Task is specified as a *4-tuple* of: *"Task ID, Processing Constraints, WSL Filename, Transformation Scheme Filename"*.

Transformation Task:

< Task ID, Processing Constraints, WSL Filename, Transformation Scheme Filename >

To support a successful achievement of the overall reengineering aim, each transformation task has to have the following mandatory attributes:

- **Task ID:** To identify a transformation task or sub-task, each one has a unique ID. This ID is either automatically generated and assigned during the analysis process or the maintainer can manually assign one.
- **Processing Constraints:** In the domain of parallel computing each parallel transformation task can be underpinned or limited by the definition of parallel processing constraints. Similar to the definition of constraints to fulfil a reengineering aim within a transformation scheme description, the definition of a parallel processing constraint can be utilised to limit the overall computing time or the processing aim. They can be also deployed to describe a specific parallel processing behaviour. For example the use of only "4" computing nodes for a transformation task.
- **WSL Filename:** Within a program transformation process, the assignment of a program source is mandatory. This file represents the starting source "P_0" and refers to the WSL program to be reengineered.
- **Transformation Scheme Filename:** As described in Chapter 3.3, the development of a transformation scheme description in combination with constraints can lead to a successful computation of a transformation task. To transfer this theory to a parallel transformations processing environment, it is mandatory to outline and assign transformation schemes to a parallel transformation process. Schemes are decomposed and mapped to the parallel transformations processing environment according to specified rules. The specified file includes the transformation scheme description.

4.6.2 PTTDL's Lexical Components

The lexical components of the specified Parallel Transformation Task Description Language (PTTDL) are keywords, symbols names and literals. This section specifies the keywords, symbols, strings and ASCII characters used within the defined language. The keywords and names within PTTDL must begin with an alphabetic character. Names consist of a sequence of alphanumeric characters where there are no length restrictions.

The PTTDL is sensitive to the case of names. All keywords are upper case (e.g. PAR), with the possibility of digits. All keywords are reserved and may not be used by the maintainer for other purposes.

- **PAR:** The introduction of the "*PAR*" keyword introduces the parallelisation of one or more specified transformation tasks. The results of this is that the PTTDL compiler decomposes each specified transformation task on the basis of its attributes.
- **PLACED PAR:** The introduction of the "*PLACED PAR*" keyword introduces the assignment of one or more transformation tasks to specified computing nodes.

The symbols within the PTTDL are specified by one or two ASCII characters.

- **"?":** Leaves the choice of computation to the headnodes analysing system.
- **"%":** States a percentage symbol.
- **"-":** States a minus symbol.
- **"<":** Symbol introducing a transformation task specific attribute.
- **">":** Symbol to close a transformation task specific attribute.
- **":=":** Assignment symbol.
- **"|":** A vertical bar represents an OR operation, and represents an alternative of operations.

4.6.3 PTTDL's Syntactic and Syntax

The syntax of the Parallel Transformation Task Description Language (PTTDL) is specified within a simple metalanguage close to a Backus-Naur Form (BNF) specification. The syntax is specified in terms of a set of productions, where each deals with the meaning of syntactic categories (or non terminal symbols); and in terms of a sequence composed of literal language constructs (terminal symbols) and possible iterative syntactic categories. Names used for the syntactic specifications are lower case letters. The following example shows the syntax of the syntactic category WSL Filename:

WSL Filename = <WSL Filename>

This productive means the "WSL Filename specifies the WSL Filename for a program transformation process" .

The vertical bar "(|)" means "OR". As an example, Computing Node specifies a "Computing Node":

```
Computing Node      = Compute Node
                    | ?
                    | IP Address
                    | DNS Node Name
```

is the same as:

```
Computing Node      = Compute Node
Computing Node      = ?
Computing Node      = IP Address
Computing Node      = DNS Node Name
```

The written structure of PTTDL constructs is specified by its syntax. Each statement in this language occupies a single line, and the indention of each statement forms a part of the syntax of the language. The following example shows the syntax for the *"par"* specification according to:

```
par                 = PAR
                        { par }
                    | Transformation Task
```

The syntax expresses *"par"* as the keyword *"PAR"* followed by zero or more processes (transformation tasks), each one on a separate line and two spaces beyond *"PAR"*. Instead of BNF's recursive definitions, curly brackets are used to indicate that a syntactic object may occur a number of times par. It is important to note that spaces at the start of the line indicate the structure of the language. Syntactic rules must always be considered in conjunction with relevant semantic rules which are given informally in words. For better readability additional spaces may be added between lexical units. Syntactic rules must always be considered in conjunction with relevant semantic rules which are given informally in words, following each group of productions. In other words, the specification of *"WSL Filename"* is qualified by a semantic rule stating that the data/specification of the expression assigned must always be the same.

4.6.4 PTTDL's Semantic Rules

After the characterisation and specification of parallel transformation tasks, their attributes and lexical units, a parallel process and therefore a computation of transformation tasks and parallel transformation tasks can be further refined and outlined. The basis of this was the syntax and semantic rules of the Occam programming language [62]. The PTTDL's syntax and program format is specified as follows. Once its basic syntactical objects and semantic rules are specified, their semantic meaning is determined as:

parallel	= Parallelisation | par | placedpar	(1)
par	= PAR { par } | **Transformation Task**	(2)
placedpar	= PLACED PAR { placedpar } | Computing Node Protocol (**Transformation Task**)	(3)
Computing Node	= Compute Node | ? | IP Address | DNS Node Name	(4)
Protocol	= Communication Protocol | ? | RMI | TCP/IP	(5)
Task ID	= number | ? | integer	(6)

Processing Constraints	= value | Parallel Processing Architecture | Speed-Up | FermaT Path | Quantifier Grain Factor | Parallel Processing Constraints	(7)
Parallel Processing Architecture	= value | "Archi:" Cluster | "Archi:" Linear Line	(8)
Speed-Up	= number | "SU:" ? | "SU:" integer "%"	(8)
FermaT Path	= number | "FP:" integer | "FP:" integer "-"integer	(9)
Quantifier Grain Factor	= number | "GF:" ? | "GF:" integer	(10)
Cluster	= Number of Compute Nodes | "CNS:" ? | "CNS:" integer	(11)
Linear Line	= Lines of Compute Nodes | "LA:" ? | "LA:" integer "x" ? | "LA:" integer "x" integer	(12)
WSL Filename	= <WSL Filename>	(13)

Transformation Scheme Filename = <Transformation Scheme Filename> (14)

Once the basic syntactical objects and their semantic rules are specified, their semantic meaning is defined as:

- **(1):** Parallel transformation tasks specified by this language are either introduced via the "*PAR*" or "*PLACED PAR"* keyword. The syntactic lower-case units "*par*" or *placedpar* further outline these two parallelisation processes. The "*PAR*" stands for the parallelisation of a transformation task or tasks by which the headnode evalu ates a parallel processing roadmap whereas the "*PLACED PAR*" keyword introduces the assignment of a transformation task to a specific computing node.

- **(2):** The syntactic lower-case unit "*par*" which specifies the keyword "*PAR*" introduces the parallelisation of zero, one or more transformation tasks based on the language syntactical definition. Once the "*PAR*" keyword is introduced any transformation task can be further refined according to its description specified by its attributes: "< *Task ID, Processing Constraints, WSL Filename, Transformation Scheme Filename* >". By utilising this construction more than one transformation task can be outlined and specified. Once assigned, the headnodes analyser performs tasks analysis and parallelises them according to their constraint definition.

- **(3):** The syntactic lower-case unit "*placedpar*" which specifies the keyword "*PLACED PAR*" introduces the assignment of zero, one or more transformation tasks to a specific computing node. The "*PLACED PAR*" construction is utilised to directly assign transformation tasks to a specific computing node. The task specific computing node and the protocol attribute are prefixed according to the task description language. This construction is also internally utilised by the headnode to directly specify which transformation task should be computed by which computing node.

- **(4):** A Computing Node can be either specified by a "?" an Internet Protocol (IP) address or a Domain Name Service (DNS) node name. A question mark specifies, that the transformation task computing node assignment should be evaluated by the headnode. The parallel system automatically knows which computing nodes are available. The second option is where a computing node IP address is directly specified or the DNS node name is stated.

- **(5):** For the task specific assignment, three network communication Protocols are available. A question mark ("?") leaves the choice / decision of the protocol to

the headnode whereas the specification of Remote Method Invocation (RMI) or Transmission Control Protocol (TCP)/IP communication, defines the usage of one or the other.

- **(6):** The Task ID of a transformation task can be either directly assigned through a value or by a question mark ("?"), which leaves the choice to the headnode.

- **(7):** Processing Constraints to further refine the parallel processing behaviour of an assigned transformation task can be either: the Parallel Transformation Processing Architecture, Speed-Up, the FermaT Path or a Quantifier Grain Factor. To further speed-up or refine a transformation task, the maintainer has the option to combine these constraints.

- **(8):** The parallel processing architecture which should be utilised for each specified transformation task can be outlined. The task description offers two options for "Archi:": Cluster or Linear Line processing. Based on these values, one or the other is utilised and can be further specified as illustrated.

- **(9):** The task specific *"Speed-Up"* which should be at least achieved through parallel computation can either be evaluated by the headnode or specified through a stated value. The specified value of "x" (SU: x %) will always result in an acknowledgement to the maintainer, confirming if the specified speed-up can be fulfilled by the parallel processing environment or not.

- **(10):** Knowing that the transformation search-space of transformation scheme descriptions and tasks can be enormous, a specified "*FermaT Path*" limits the parallel process search-space and could result in further speed-up. However it has to be kept in mind, that the reengineering constraints may not be fulfilled. This attribute is also utilised by the headnode to further express on which AST path of the corresponding WSL program the task specific computing node should start its parallel processing. Values can be expressed as "FP:3" or "FP:1-3" which states that a transformation should be applied on FermaT tree path "3" or in between FermaT path "1-3".

- **(11):** The "*Quantifier Grain Factor*" value of "x" (GF: x), can be utilised to further specify the parallel decomposition of a quantifier-construct within a transformation scheme description. For example, the value "GF:2" should divide the quantifier interval "[1-4]" into two individuals "[1-2]" and "[3-4]".

- **(12):** The "*Cluster Architecture*" attribute can further specify the parallel processing layout utilised. It can specify the number of computing nodes which should

be involved within a parallel transformation process or it can also limit the involved computing nodes. This leaves more computing resources for other computing tasks. By stating a "?", the choice of the best appropriate solution is decided by the headnode.

- **(13):** The design of a "*Linear Line*" parallel transformation processing architecture can either be limited to a one-dimensional or to a two-dimensional linear line array or left completely open for the definition by the headnode, expressed via "?". "LA: 3x3" would create a linear line array of 3 x 3 computing nodes, which would include 9 computing nodes to compute the transformation task.

- **(14):** Specifies the "*WSL Filename WSL* P_0" which should be transformed within the transformation task. Based on the parallel architecture utilised, Cluster or Linear Array, more than WSL program file can be specified within a parallel transformation task description.

- **(15):** Specifies the "*Transformation Scheme Description Filename*" which should be taken into consideration for the parallel computation of the specified transformation task.

4.6.5 Parallel Transformation Task Example

Based on these semantic definitions, independent parallel transformation tasks can be specified and described via the presented language. Even further, this specific technique opens the possibility to totally automate parallel computations. Utilising a queuing system, more than one transformation task can be specified and submitted to the parallel environment. The following highlights the defined syntax and semantics in more detail.

Nevertheless it has to be distinguished between the assignment of a transformation task to the parallel processing environment or the assignment to a specific computing node. Both constructs are combined within the same language. The later is also used by the headnode to internally save and assign transformation tasks to computing nodes. Once defined, it has to be distinguished between the "*PAR*" and the "*PLACED PAR*" construct. The expressiveness of the "*PAR*" construct is illustrated in Listing 4.1 by generally introducing the parallelisation of transformation tasks, whereas the "*PLACED PAR*" construction in Listing 4.2 expresses a specific assignment of a transformation task to a computing node.

Nevertheless it has to be distinguished between the assignment of a transformation task to the parallel processing environment or the assignment to a specific computing node. Both constructs are combined within the same language. The later is also used by

the headnode to internally save and assign transformation tasks to computing nodes. As defined it has to be distinguished between the "*PAR*" and the "*PLACED PAR*" construct. The expressiveness of the "*PAR*" construct is illustrated in Listing 4.1 by generally introducing the parallelisation of transformation tasks, whereas the "*PLACED PAR*" construction in Listing 4.2 expresses a specific assignment of a transformation task to a computing node.

```
1 PAR
2   (Task ID, Constraints, WSL Filename, Transformation Scheme Filename)
3   (Task ID, Constraints, WSL Filename, Transformation Scheme Filename)
```

Listing 4.1: PAR: Parallel Transformation Task Construct

As demonstrated, the "*PAR*" construct follows a specific transformation task description based on the developed syntax and semantics, whereas the "*PLACED PAR*" can be considered as a refinement or decomposition of a transformation task. Both task description constructs are saved as Parallel Transformation Task Description (PTTD) files before the parallel computation starts.

```
1 PLACED PAR
2   IP Protocol
3         (Task ID, Constraints, WSL Filename, Transformation Scheme Filename)
4   IP Protocol
5         (Task ID, Constraints, WSL Filename, Transformation Scheme Filename)
```

Listing 4.2: PLACED PAR: Parallel Transformation Task Construct

Listing 4.3 gives an illustration on how a PTTD file with two transformation tasks is outlined. The first parallel task is defined as follows: Task ID is "1", followed by the parallel processing constraints *"Archi:Cluster:CNS:4"*: specifying the usage of a cluster architecture and 4 computing nodes for parallel computation. "SU:100%" specifies an at least 100% achievable speed-up through parallel computation. The WSL program file to be reengineered is specified within filename *"example.wsl"* and the transformation scheme description is specified in *"example.tdsl"*. Within the second transformation task specification, the maintainer left the cluster specific parallel transformation processing solution totally open to the headnode's computation. This results in a headnode action to choose a suitable parallel processing roadmap and maximum performance. The question mark "?" symbolises this behaviour.

```
1 PAR
2   (1, Archi:Cluster:CNS:4 SU:100%, example.wsl, example.tdsl)
3   (2, Archi:Cluster:?, example2.wsl, example.tdsl)
```

Listing 4.3: PAR: Cluster Parallel Transformation Task Construct

In comparison to above, Listing 4.4 demonstrates how a Linear Array transformation processing architecture for parallel computation can be described. This technique is further discussed in Chapter 7, but the main intention of this specific technique is the decomposition of transformation sequences and individual transformation assignments to specific computing nodes. Following this parallel processing scheme more than one WSL program file can be processed at a time. The difference between the above stated cluster technique is not only the alignment of the computing nodes and their computation, it also results with a minimum time delay between the computation of different WSL files. The second specified file is usually immediately processed after the task specific computing node has performed its transformation on the first WSL program file.

```
1 PAR
2   (3, Archi:LA:3x3, example.wsl example2.wsl, example.tdsl)
```

Listing 4.4: PAR: Linear Line Parallel Transformation Task Construct

In comparison to the "*PAR*" construct, the "*PLACED PAR*" construct not only assigns transformation tasks directly to computing nodes, it can be also considered as the last pre-processing/analysing step before the parallel computation starts. Listing 4.5 gives an example. The first task description describes the assignment of transformation task number "1" to a computing node with the IP "*192.168.50.4*" and suggests the usage of Remote Method Invocation (RMI) as a network communication protocol. Whereas in the second example, task number "2" is assigned to the computing node with the domain name "*fc-node3.homeip.net*" utilising TCP/IP as a communication infrastructure. In addition each transformation task is underpinned with their WSL program- and transformation scheme description file names, in this case "*example.wsl*", "*example.tdsl*" and "*example2.wsl*", "*example2.tdsl*".

```
1 PLACED PAR
2   192.168.50.4 RMI (1, example.wsl, example.tdsl)
3   fc-node3.homeip.net TCP/IP (2, example2.wsl, example2.tdsl)
```

Listing 4.5: PLACED PAR Construct

In a nutshell, by introducing this parallel transformation task description language the following advantages are identified:

- A formal language to describe and outline parallel transformations processes is presented.
- Presentation of automatic parallel transformations processing facilities through maintainer specificParallel Transformation Task Description (PTTD) files.
- A "*PLACED PAR*" construct to identify if pre-processing steps are fulfilled.
- "*PLACED PAR*" constructs can be re-used to write direct parallel processing assignments similar to batch file processing.
- Presentation of a structured approach of parallel transformation processing.

4.7 Parallel Transformations Framework Analysing System

The analysing system which has been developed for the presented parallel transformations framework, to analyse transformation tasks before the actual transformation process starts, can be considered as one of the system key elements. Similar to pre-processor techniques utilised in programming languages as C, the analyser evaluates input-data and produces output-data. Figure 4.3 gives the architectural overview of the developed system. The system is further outlined in Chapter 5 and distinguishes between different pre-processing modes: **fully automatic** and **maintainer mode**, guided mode.

Within fully automatic mode, the headnodes analyser automatically reads and evaluates parallel transformation tasks (input-data) and produces parallel transformations processing outlines (output-data). Transformation task definitions can include specified parallel processing behaviour. Input-data consists of transformation processing information combined to a parallel transformation task description. This includes the number of computing nodes involved, the WSL code files which should be processed in parallel and the outline of the transformation process defined as a transformation scheme description. After the task assignment, the parallel transformations system runs on its own and produces the output-data. Depending on the task specification, the system runs until all information is processed or stops with a failure warning. In a case of a failure they are recorded and could include a computing node failure or a failure during a transformation process.

In this case the system switches to the maintainer mode, and the maintainer is given the possibility to describe and assign transformation tasks manually to the parallel transformations processing system. This process can be performed by utilising the Graphical

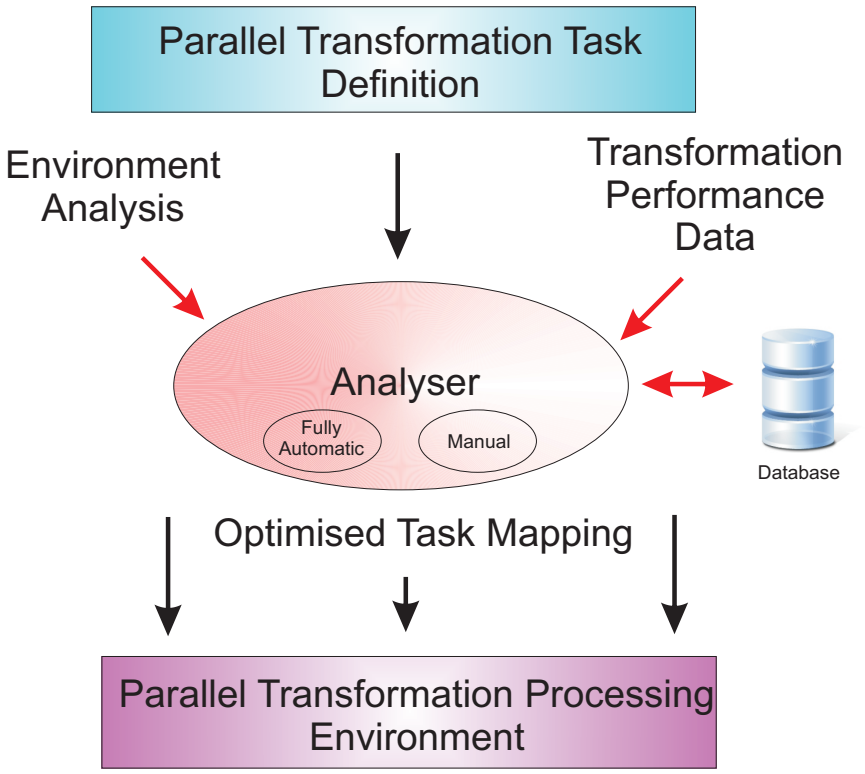


FIGURE 4.3: Parallel Transformations Framework: Analyser Model

User Interface (GUI) developed and outlined in Chapter 8. With this tool and the refinement of parallel transformation tasks, the user can describe and outline parallel transformations jobs. Embedded transformation scheme descriptions allow the maintainer to outline which WSL files should be processed and in which order they should be computed.

Once computing nodes are registered within the parallel transformation processing environment, parallel transformation tasks can be evaluated and computed. In both preprocessing modes the analysing system evaluates a suitable parallel processing solution based on its task definition. How the analyser evaluates and computes based on parallel transformation tasks descriptions is presented in Chapter 5.2.2.

4.8 Parallel Transformations Processing Techniques

To be able to fulfil parallel transformations processing in regard to the FermaT transformation system, different parallel processing techniques need to be utilised. Not only that the developed transformation processing approach should be as flexible and adjustable as possible, the parallel processing solution (roadmap) produced by the developed analysing

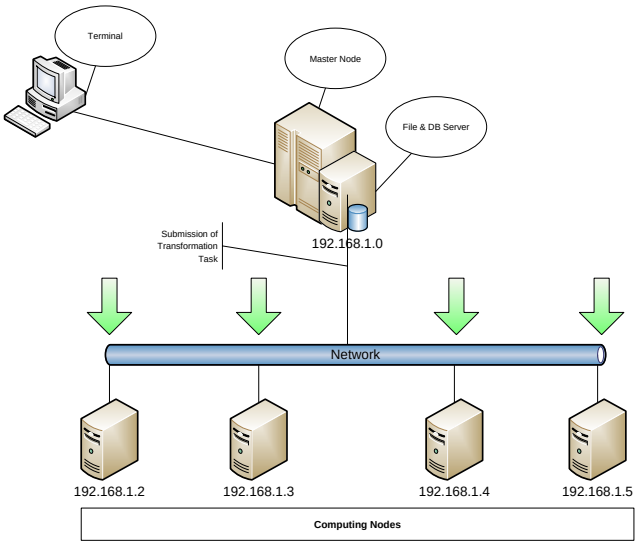
system should also be an efficient one. One aspect of this evaluation process is the defined parallel transformation task description language. The best parallel processing strategy is usually evaluated by analysing the overall transformation target embedded within each task. For the decision of the appropriate parallel processing solution, some reference transformation processing time tables are taken into consideration. The table includes the times how long it takes for each computing node to apply the specified transformations within the transformation system.

As outlined in Section 4.4, the parallel transformation process need to insure semantically equivalence. This means that the reengineered WSL program need to have the same semantically equivalence as the original program code specified at the beginning of the parallel computation. This is preserved by letting computing nodes compute/-transform independent parts of the overall transformation scheme description. More precise, independent parts of the generated transformation sequence search space. How transformation scheme descriptions are decomposed is explained in Chapter 6. Chapter 7 illustrates how independent parts are computed in parallel and results are evaluated. Based on these refinements two differently behaving parallel transformations processing modes, **Parallel Processing** and **Linear Array Processing** arose and are illustrated in Figure 4.8.

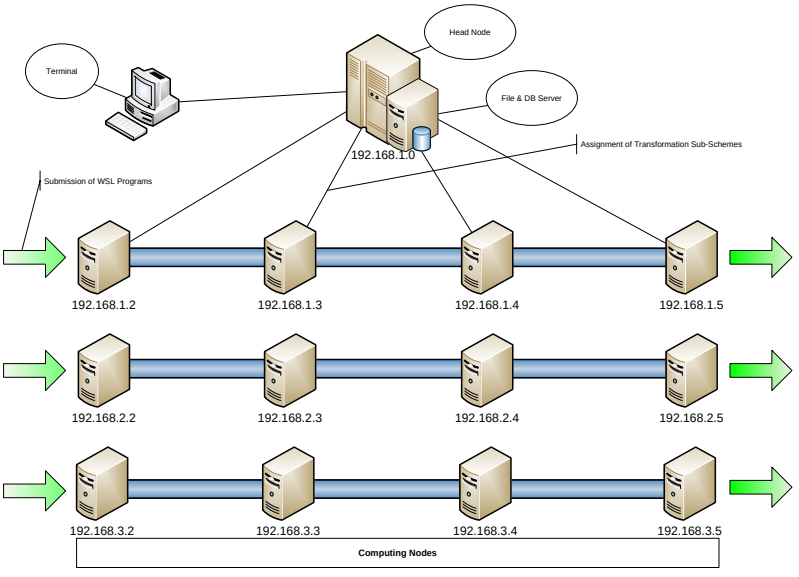
- Within the **parallel processing mode** it has to be distinguished between the fully automatic parallel processing and the manual parallel processing mode. The difference between those two modes is, within the fully-automatic parallel transformation process the maintainer only has to submit a predefined parallel transformation task, whereas in the manual-mode the maintainer manually describes and outlines a parallel transformation process utilising the developed tool support outlined in Chapter 8. Within both processing modes the parallel transformation task and transformation scheme description are evaluated, decomposed and mapped to the parallel processing architecture. The basis for this outline behaviour is embedded within the transformation task description. Depending on the number of available or specified computing nodes, each computing node only processes a part of the overall transformation task. Further details, on how this mode performs and how transformation tasks are assigned and submitted to computing nodes is outlined in Chapter 7.

- Behind the **parallel linear array processing mode** stands a different concept. The main intention of this parallel processing technique is the assignment of a transformation sub-scheme to a linear line of computing nodes. Linear line functions can be considered to be pipeline principals. This results in the assignment of transformations to individual computing nodes. The architectural-outcome of this

specific technique is a pipeline related construct and behaviour. Knowing which sub-transformation scheme description or transformations belong to which linear line of computing nodes, more than one WSL program file can be piped/transformed at a time. Details on how this mode performs and how transformation tasks are assigned and submitted to computing nodes can be found in Chapter 7.



(a) Parallel Processing



(b) Linear Array Processing

FIGURE 4.4: Parallel Transformations Processing Techniques

These parallel transformations processing techniques should outline that different parallel processing techniques are utilised and adapted for the established parallel environment. Moreover, the main intention of this approach is the achievement of a dynamic and flexible environment in regard of parallelisation of transformation tasks. Chapter 9 discusses how these techniques perform under realistic conditions.

4.9 Communication System for Parallel Transformations Processing

Within every parallel computing environment, suitable parallel processing communication techniques have to be utilised. Similar to the de facto Message-Passing-Interface (MPI) standard commonly used to communicate between different parallel processes within Symmetric-Multi-Processing (SMP) systems [63], a special transformation processing message-passing system has been developed for this approach. To directly assign transformation tasks to computing nodes, a formal language based on Occam [62] has been defined to adapt other parallel computing and communication capabilities. The difficulty has been identified in implementing some commonly used MPI standards within the FermaT transformation system. By the identification of the key features described in Section 4.2, the attributes stated below on this mainly computation based parallel transformations processing environment are taken into consideration:

- Each parallel computing component, headnode same as a computing node is equipped with a small kernel system. A message passing system has been established for communication. The communication system supports to add computing nodes during runtime to the system.
- Computing nodes communicate with the headnode via message-passing.
- The possibility to directly assign transformation tasks to computing nodes.
- Different parallel processing techniques are utilised to perform parallel transformation tasks. Therefore the developed communication system needs to be flexible while asynchronous and synchronous communication between the computing nodes should be preserved and possible.

To illustrate how parallel transformations processing can differ in speed in its simplest form is demonstrated by utilising open-source software components. Two parallel communicating systems are implemented and tested within this approach. Both are based on the Operating System (OS) TCP/IP protocol stack. One consists of the basic TCP/IP

communication infrastructure, the other utilises Remote Procedure Calls (RPCs). These systems only serve as comparison. Further explanation on how the developed communication system has been set-up is explained in Chapter 5.2.6.

4.10 Summary

This chapter gives an insight into the architecture and the basic concepts behind the proposed parallel transformations framework. It presents its key features and illustrates how a cluster computing system can be utilised for parallel program transformations processing. It presents an outline of the utilised parallel processing techniques and special parallel features. A parallel transformation task description language which can be utilised by maintainer to specify and assign parallel tasks to the proposed environment has been specified and examples are given.

Chapter 5

Parallel Transformations Framework: The Architecture

Objectives

- Presentation of service provided by the parallel transformations framework.
- Explanation of services within the systems headnode.
- Description and outline of transformation task application.
- Outline of services within a computing node.

5.1 Introduction

This chapter gives an overview and explanation of the architecture and services established for the proposed parallel transformations framework. The presented techniques are mainly headnode and computing node related. Each is described in their order, as this knowledge is needed to understand the procedure of the established parallel transformations processing architecture. The main focus has been on the explanation of the headnodes key feature: "*an analysing system*". This analyser performs a complete environment and transformation task analysis, to ensure successful computations. The presentation of a WSL program code analysis underpins these aims. In order to fully understand the sequence of processing steps performed by this parallel environment, the following sections are written in the order in which the processing steps are performed.

5.2 The Headnode Services

As outlined within the design specification of the parallel transformations system described in the previous chapter, most of the parallel systems knowledge is combined within an analysing module. This component runs inside the headnode and collects as much information as possible from the other modules. Figure 5.1 gives an overview of the headnode services.

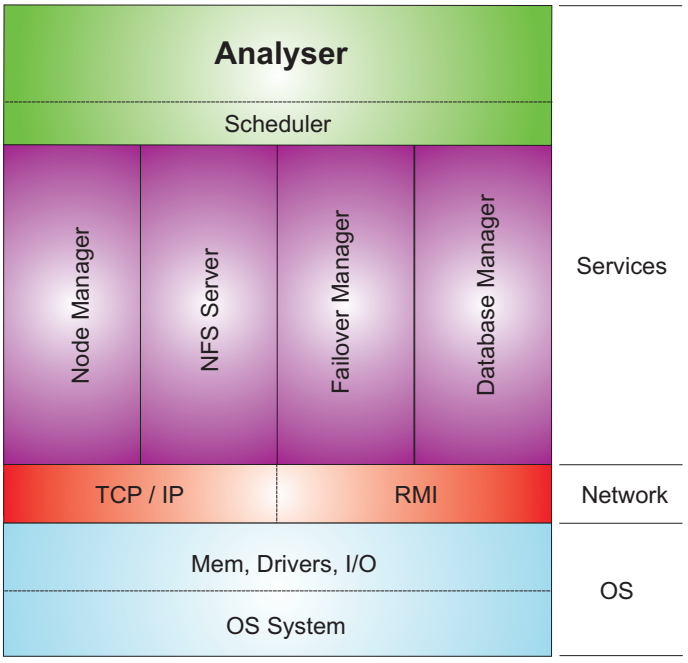


FIGURE 5.1: The Parallel Transformations Systems Headnode Services

The headnode's main services are categorised as the following: Computing Node-Manager, Network File System (NFS)-Manager, Failover-Manager and Database-Manager. All services are mostly controlled by the analyser module which can be considered as a preprocessing component. Before a parallel transformation task is computed, this analyser performs a precise parallel transformation task analysis based on specified parameter. Each service module works independently and all act as separate instances. The result is, they respond very quickly to changing environment circumstances.

5.2.1 Node Manager and Environment Analysis

The headnode's Node Manager is a service utilised by computing nodes to register their services. Computing node services are either transformation or evaluation based. The node management's only purpose is to evaluate how many computing nodes are

available within the parallel system and of which type they are. As explained in the sections of the analysing system, these values are needed to calculate a weighted and balanced parallel processing schedule based on parallel transformation task specification, generated transformation sequences and transformation sub-schemes. This service is also utilised to record and interpret computing node information which also includes their current processing status. Their status is usually transmitted via a messaging service explained in Section 5.2.6. During the work node registration process, the following data structure is recorded and saved for each node, unless this information is already known and available within the system:

- Central Processing Unit (CPU) type and processing speed.
- Random Access Memory (RAM) size.
- Available Hard Disk Drive (HDD) space.
- FermaT transformation processing time table.

During the computing node registration process, a specified computing node FermaT Performance Test usually evaluates these values. Unless the node hardware changes or the system maintainer likes to update these values, they are used for each parallel transformation task analysis process. More information about this test can be found in Chapter 8. The only purpose of gathering this information from all computing nodes is to provide the headnode with a clear view of available services and system performance.

5.2.2 The Analysing System

As discussed in Chapter 4.2 about the key-features of this proposed parallel transformations framework, the system's analyser plays an important role in the process of dynamic parallel transformations processing. The analyser can be considered as the central unit between the presented headnode services, as explained and illustrated in Figure 5.1. The system tries to gather as much information as possible, to calculate suitable parallel transformations processing outlines. The specified transformation tasks are the basis for this calculation. To recapitulate, the following steps are performed repetitively by the analysing system for each task:

1. Analysis of the parallel transformation task followed by:
    - Examination of the parallel processing environment.
    - Evaluation of transformation processing time.

- Analysis of the transformation scheme description file.
- Evaluation of the WSL program file "P_0" to be reengineered.

2. Decomposition of the transformation scheme description based on decomposition laws.

3. Calculation of a parallel transformation processing roadmap based on:

   - Evaluated information.
   - Possible parallel processing constraints specifying:
     - Number of computing nodes involved.
     - Parallel processing technique.
     - Specified reengineering aim.
   - Parallelisation possibilities within the transformation scheme or sub-schemes.

4. Distribution, scheduling and evaluation of transformation tasks.

The headnode's analyser always tries to apply this pattern on every transformation task. As this is an automated process, the headnode automatically tries to find a suitable parallel processing technique according to the task definition. As mentioned previously, these steps can be also assigned and outlined manually. Processing Step "*1*" is outlined and described in detail in the following sections, as the involved steps are based on the services developed and explained within this chapter. Processing Step "*2*", the decomposition and parallel analysis of transformation scheme descriptions is presented in Chapter 6. The calculation of parallel transformations processes is also highlighted in the same chapter. The following example gives an illustration on how the developed analyser parallelises transformation data structures and produces appropriate processing solutions.

```
1 PAR
2   (101, Archi:Cluster:CNS:2 SU:100\%, hello_world.wsl, hello_world.tsdl)
```

Listing 5.1: Parallel Transformation Task Definition

Listing 5.1 presents a parallel transformation task defined by utilising the developed task description language described in Chapter 4.6. The task specific parameters are: task ID is "*101*", followed by the specification of two parallel processing constraints of: 1) the use of a cluster architecture for parallel computing utilising "*2*" computing nodes; 2) trying to achieve an overall task speed-up of at least 100%. The initial WSL program source ("P_0") is specified in file "*hello world.wsl*" and the corresponding task

specific transformation scheme description is outlined in file "*hello world.tsdl*". Within the next processing step, the within the task specified WSL program file presented in Listing 5.2, is loaded into the headnodes transformation engine. Within FermaT, each WSL program source is represented as an Abstract Syntax Tree (AST). The AST of the specified program source is illustrated in Figure 5.2.

```
1 IF x = 0 THEN PRINT("Goodby cruel world")
2 ELSIF FALSE THEN PRINT("Goodby cruel world")
3 ELSIF TRUE THEN PRINT("Hello world")
4 ELSE y := 2 FI
```

Listing 5.2: WSL Program: Hello World

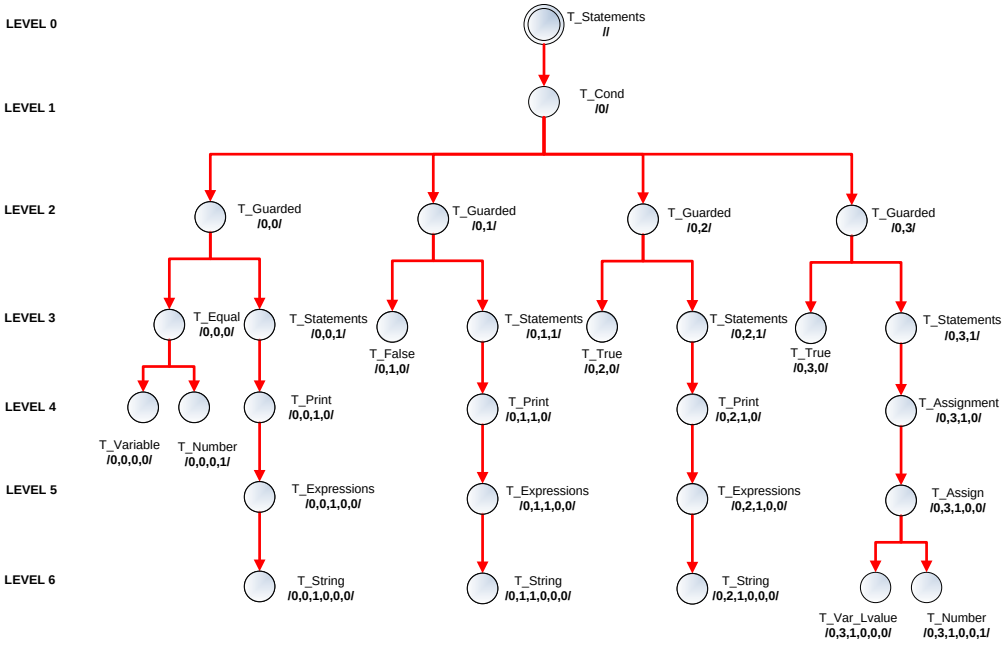


Figure 5.2: WSL AST Code Analysis: Hello World

The next pre-processing step includes the evaluation of which WSL specific AST types belong to which AST path within the specified WSL program. To do so, the analyser searches the complete WSL program tree and creates indices as presented in Table 5.1. This results in a first-time WSL program source analysis and opens more possibilities for parallelism. The utilised searching technique is based on the standard breadth-first search algorithm, which starts at the top of the AST with the root node, in this case AST node "(T_Statements) //" and searches its way down. Within the given example, the different search levels are cited on the left side, as presented in Figure 5.2. The

search-depth of WSL programs is not very deep, compared to the Lines Of Code (LOC) of program sources. The utilised searching algorithm seems to be the most appropriate one, because it can also be parallelised, in cases of longer and more detailed parallel transformation processing analysis. Larger WSL translated assembler modules usually expand in size during the FermaT translation process. The reason for this is because the FermaT transformation system captures as much information as possible from the starting program source files. These searching techniques need longer to perform and the algorithms could be parallelised.

TABLE 5.1: AST Type FermaT Path Relation: Hello World

AST Type	AST Paths
T_Statements	//, /0,0,1/, /0,1,1/, /0,2,1/, /0,3,1/
T_Cond	/0/
T_Guarded	/0,0/, /0,1/, /0,2/, /0,3/
T_Equal	/0,0,0/
T_True	/0,2,0/, /0,3,0/
T_Variable	/0,0,0,0/
T_Number	/0,0,0,1/, /0,3,1,0,0,1/
T_False	/0,1,0/,
T_Print	/0,0,1,0/, /0,1,1,0/, /0,2,1,0/,
T_Assignment	/0,3,1,0/
T_Expressions	/0,0,1,0,0/, /0,1,1,0,0/, /0,2,1,0,0/
T_Assign	/0,3,1,0,0/
T_String	/0,0,1,0,0,0/, /0,1,1,0,0,0/, /0,2,1,0,0,0/
T_Var_Lvalue	/0,3,1,0,0,0/

The analysis of the task specific transformation scheme description, presented in Listing 5.3 , reveals that only a single FermaT transformation should be applied on the presented WSL program code. In addition to the transformation, an extra AST tree node is specified, stating on which tree leaf the transformation should be applied. It is of the WSL type "*T Cond*". To evaluate on which program tree path or paths the specified transformation is applicable, the pre-processed AST type path table is utilised and states that AST type "*T_Cond*" was found on AST path "/0/". This is the only location on which the transformation can be applied. Within a FermaT transformation process, an AST path is needed to tell the transformation engine on which program branch a transformation should be applied.

```
1 {
2   <Simplify If @ T_Cond>
3 }
```

LISTING 5.3: Transformation Scheme Description: Hello World

Further analysis of the parallel transformation task description reveals that another specified parallel processing constraint is *"SU:100%"*. This specifies that an overall speed-up of at least 100% should be achieved. To demonstrate this procedure within the given example, the WSL program is extended in size by doubling its WSL program constructs. This results in a duplication of all AST types and ends in having two WSL AST types "*T Cond*" within the program search space. This opens the possibility to apply the transformation "*Simplify If*" twice on two different FermaT "*IF*" statement paths of the AST type "*T Cond*". Listing 5.4 presents this example, which could be utilised to demonstrate the specified parallel transformations processes.

```
1 IF x = 0 THEN PRINT("Goodby cruel world")
2  ELSIF FALSE THEN PRINT("Goodby cruel world")
3  ELSIF TRUE THEN PRINT("Hello world")
4 ELSE y := 2 FI;
5 IF x = 0 THEN PRINT("Goodby cruel world")
6  ELSIF FALSE THEN PRINT("Goodby cruel world")
7  ELSIF TRUE THEN PRINT("Hello world")
8 ELSE y := 2 FI
```

LISTING 5.4: Extended WSL Program: Hello World

Once all this information has been evaluated, the transformation task can be parallelised. How the developed parallel transformation processing technique produces a suitable parallel processing roadmap is explained in Chapter 7.

5.2.3 A Network File System (NFS) to Access Processing Data

Each computation, in which computing nodes are involved, needs to have access to data produced by the headnode. The utilisation of a Network File System (NFS) directory seems to be an appropriate solution to resolve this. The technique opens the possibility for each computing node to read and write processing data directly to a location, which is accessible to all components involved during parallel transformations processes. The NFS is usually part of every Linux Operating System (OS) and is therefore accessible once the parallel transformations processing system has been set-up and started. Server side configuration changes can be directly made within the headnode, outlined within the prototype tool support section. The system's NFS concept is presented in Figure 5.3.

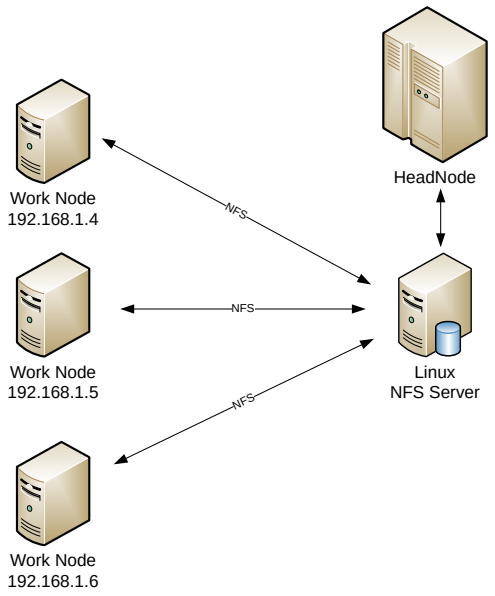


FIGURE 5.3: The Parallel Transformations System NFS Service

To start the calculation and computation of parallel transformation tasks, the headnode has to have access to a specified NFS directory, to read and evaluate the following files:

- Parallel Transformation Task Description (PTTD) file "Tk_0".
- Initial WSL program source file "P_0".
- Transformation Scheme Description Language (TSDL) file "T_0".

After the completion of the pre-processing steps performed by the headnodes analysing system, the following parallel transformation task specific files will be placed within the same NFS directory, readable by all computing nodes:

- Transformation Task Description (TTD) file for each computing node "Tk_0 - Tk_n".
- Initial WSL starting program file "P_0".
- Transformation task specific TSDL sub-files "T_0 - T_n".

Within the next processing step, the transformation tasks are distributed among the computing nodes according to its task and sub-task specification. How compute nodes access this data and start their transformation processing steps is discussed in Section 8.

5.2.4 The Database Service

To store analysing and parallel transformation processing data as knowledge for current and upcoming transformation tasks, a Database-Service has been designed. This service records all service states illustrated: transformation scheme descriptions and sub-scheme information, transformation time tables, transformation application checks and capabilities followed by computing node info. By establishing such a service, parallel transformation task specific file information or already evaluated knowledge does not need to be created, analysed or parsed if additional processing information is needed, has been lost, or the headnode has been restarted. Also during a computing node failure gathered or already assigned information can be easily recalculated. The following outlines the most commonly used database entries. Since each parallel transformation task outlined within a Parallel Transformation Task Description (PTTD) file, can be identified by a unique value id *"PaTransTask ID"*, this value is saved for further task evaluation purposes within the database.

Since the PTTD file describes and specifies a parallel transformation task, the same information is saved as an entity within the database. Utilising this technique avoids time-consuming and repetitive parsing processes of parallel transformation processing files during headnode restart or pre-processing failure.

```
1 < PaTransTaskID > < PaPingCons > < WSLFile > < TSDnFile > < NoCNS > < Info > <
    PTTD Filename >
```

Listing 5.5: PaTransTaskID Description

After the overall specification of the parallel transformation task has been subdivided into independent parallel sub-tasks, their associations *(TransTaskID)* and the corresponding transformation task information are saved as associated entities within the database, listed in Listing 5.6. As during the analysis performed by the headnode the initial transformation scheme description is decomposed into sub-schemes and assigned to independent transformation sub-task, their associations are also saved within the database and separate sub-schemes description files. All attributes specified are stored as an entity of: transformation task ID *(TransTaskID)*, computing node IP address *(CnIP)*, corresponding constraint *(PingCons)*, WSL file *(WSLFile P_i)*, transformation sub-scheme definition file *(TSDnSubFile T_n)* and some additional information.

```
1 < TransTaskID-1 > < CnIP > < PingCons > < WSLFile > < TSDnSubFile-1 > < Info >
2 < TransTaskID-2 > < CnIP > < PingCons > < WSLFile > < TSDnSubFile-2 > < Info >
3 < TransTaskID-3 > < CnIP > < PingCons > < WSLFile > < TSDnSubFile-3 > < Info >
4 < TransTaskID-4 > < CnIP > < PingCons > < WSLFile > < TSDnSubFile-4 > < Info >
5 < TransTaskID-...> < CnIP > < PingCons > < WSLFile > < TSDnSubFile-... > < Info >
```

Listing 5.6: Transformation Sub-Task Descriptions

Having specified these parallel transformation processing database tables, current and future parallel transformation tasks can be assisted by recalling or reusing already collected information.

5.2.5 The Transformation Task Recovery-Service

The transformation task recovery-service developed for this parallel transformation processing architecture is closely related to the techniques used in High-Availability (HA) clusters explained in Chapter 2.7.2. To ensure that computing node failure is captured and its processes are successfully restored during parallel transformations processing, the utilised communication system needed to be extended. To provide such a facility the proposed communication messaging service is enlarged to send and receive heartbeat signals from computing nodes in a predefined interval. This rhythm can be adjusted through the developed support tool. To track and record all processing steps, the recovery-service interacts very closely with the developed scheduling-service. Since all transformation tasks are recorded within the transformation task bank as Transformation Task Description (TTD) files, they can be recovered during a headnode or computing node failure. Utilising the presented database technique can reactivate task information. To transfer transformation processing data to other computing nodes, the NFS directory service is utilised. Figure 5.4 gives an illustration of the developed recovery architecture.

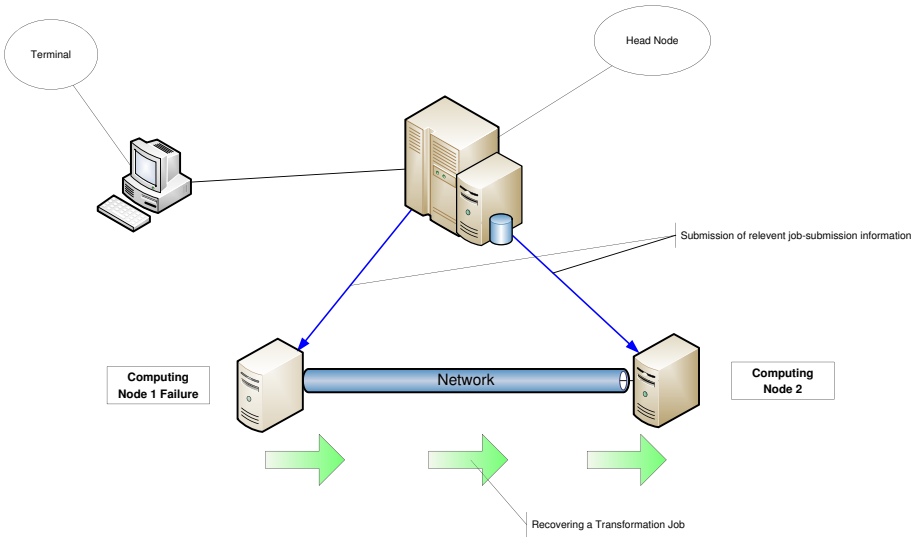


FIGURE 5.4: Recovery-Service

As an example, within "*Computing Node 1*" in Figure 5.4 a failure occurs during a transformation process. The headnode registers this by not receiving the heartbeat of the computing node within the specified time frame. Within the next processing step the headnode automatically tries to recover the computing nodes transformation engine through special developed scripts. This is performed via a SSH tunnel further outlined in Chapter 8. If this does not lead to a suitable node recovery solution, the system maintainer is notified to take a closer look at the computing node configuration. Knowing which computing node failed, through the allocation of the computing nodes IP address registered during the start-up process, the allocated transformation task can be evaluated through an entry within the processing table or database table. By evaluating this information, the specified transformation task can be reassigned to another computing node. Since each transformation task is also saved as a TTD file entity within the developed database system presented in Listing 5.7, missing pre-processing information can be also recovered by the evaluation of the computing node IP within the recorded processing table or within the headnodes parallel transformation task scheduling system.

```
1 < TransTaskID >
2       < PingCons >
3       < WSLFile >
4       < TSDnFile >
5       < CnIP >
```

LISTING 5.7: Transformation Task Description (TTD) File Database Entity

To be able to reallocate a transformation process to a stage where it stopped, a special transformation process recording technique is used. Each WSL program file on which a transformation has been processed can be identified by a unique "*filename*". This file consists of the sequence of transformations which have been so far processed on the specified file by a computing node. Furthermore an index is added, specifying on which AST tree path each transformation has been applied. As all the computing nodes processed files are either saved on its local HDD or in a specific NFS folder, the last processing stage before the failure can be reactivated. Listing 5.8 presents such a file, which starts with the transformation sequence identifier of "*SEQ112*", followed by the identification on the first transformation applied. In this case it was the transformation with the identifier "*T0*", applied on the first applicable FermaT specific path specified with a "*1*". The next utilised transformation was the one with number "*T1*" also applied on the first FermaT specific path. Within the next transformation processing step transformation "*T3*" was involved, but in this case was applied on the second FermaT specific AST path. The process on the evaluation, of which FermaT transformation can be applied on which AST path within a program source is further discussed in Chapter 7.2.1.

```
1 SEQ112_T01-T11-T32.wsl
```

Listing 5.8: WSL Program Transformation Processing File

5.2.6 The Communication Service

The communication infrastructure for this parallel transformations processing environment is based on the Ethernet network standard. To evaluate the difference in speed between common message-passing techniques, two of this kind are implemented and are compared within this approach. Both are based on theTCP/IP protocol, the first is a pure TCP/IP connection, the second communication implementation encapsulates the TCP/IP stack and provides Remote Procedure Call (RPC) procedures for communication. To be able to evaluate the difference in these two techniques, the headnode runs two communication channels in parallel. For their evaluation, they can be switched on and off depending which one is being used. The specified Parallel Transformation Task Description (PTTD) language also supports this feature. The established communication system works very closely with the developed scheduling system, described in Section 7.2.4. Since parallel transformation tasks are always submitted through the system scheduler, the communication system is directly addressed via the TCP/IP stack. Within computing node communication, it can be distinguished between a synchronous

and asynchronous type. During parallel processing, the headnode to computing node communication is always synchronous, as the headnode needs to be notified when a parallel transformation process has been completed or failed. During the linear array processing approach described in Chapter 8, communication could be of a synchronous or an asynchronous type. Utilising the asynchronous processing mode, caution has to be taken, as node- or transformation processing-failures are not recognised. Default communication set-up of linear-line processing is synchronous. Figure 5.5 gives an illustration of both techniques within the linear line processing mode. The processing line at the top symbolises the asynchronous communication whereas the bottom one expresses the synchronous communication type.

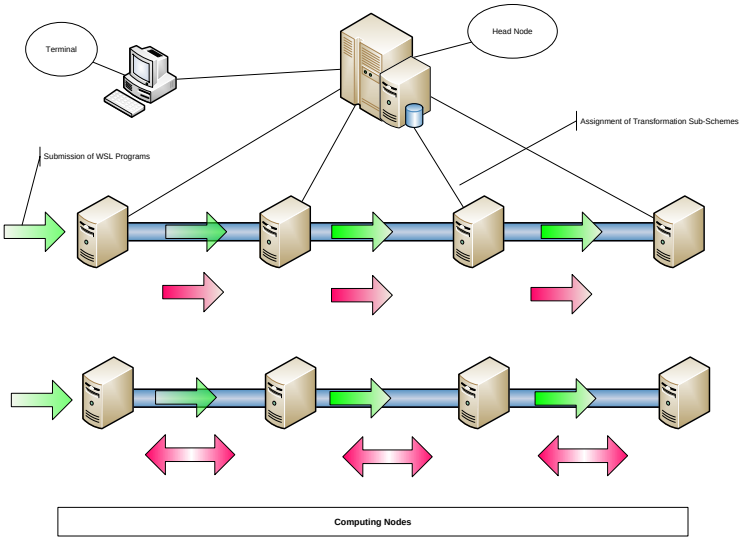


FIGURE 5.5: Linear Line Computing Node Communication

Communication Procedures

To be able to arrange the specified parallel transformations processing communication layouts on the basis of the specified protocols, specific techniques are utilised. To avoid extra communication overhead, special parallel transformations processing commands and constructs have been designed. These commands can be utilised by computing nodes to signalise computing node behaviour. Figure 5.6 gives an overview of their TCP/IP channel and data structures.

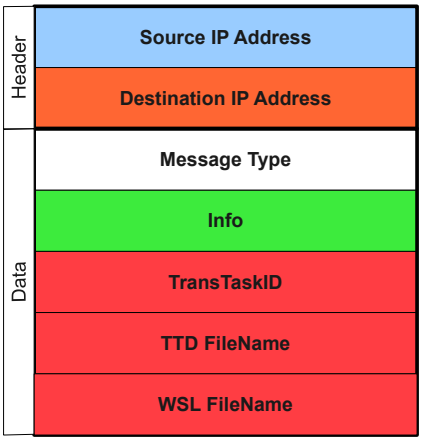


FIGURE 5.6: Parallel Transformations Processing Communication Construct

Message Type		**Data**			
	IP	**Info**	**Task ID**	**TTD File**	**WSL File**
Status	xxx	Ready / Not Ready	(xxx)		
Status	xxx	Result / Finished	(xxx)		
Failure	xxx	Reason (Log File)	xxx		
Trans JOB	xxx	Parallel	(xxx)	xxx	
Linear Line JOB	xxx	NBG1, NBG2	(xxx)	xxx	
Trans JOB	xxx	Linear Line	xxx		xxx
Found Result	xxx		xxx	(xxx)	xxx
Heartbeat	xxx	Sequence Nr	xxx		

TABLE 5.2: Communication Protocol Data Structures

The table comprises the illustrated and implemented communication methods presented and can eventually be substituted by either one of the two communication possibilities described. Both communication specifications, Remote Procedure Call (RPC) and TCP/IP, can be directly mapped to one or the other communication structure, whereas "xxx" specifies mandatory values and "(xxx)" the ones which are not mandatory.

5.3 The Computing Node Services

Similar to the headnode services explained above, each computing node is equipped with two fundamental services: a *Compute-* and an *Evaluation-Service*. Figure 5.7 presents their functions. The first processes transformations encapsulating an instance of the FermaT transformation engine as a kernel system. The second service, the evaluation

service, functions as a transformation process evaluation system. The evaluation service embeds the analysis of transformation processing results. Both run as separate instances within a computing node and work independently. This technique is required because a computing node queuing system depends on information of both modules. The reason for this is, because transformation tasks are sometimes based on the same WSL program file or parallel transformation tasks are forwarded to another computing node.

Since parallel tasks are identified by a unique "*ID*", the system can replicate them during computing node failure or information loss. Consequently this number can vary because these numbers depend on how many sub-schemes the original stated transformation scheme description has been subdivided into. To give an example, if the parallel transformation task *"ID"* is *"3"* and the transformation scheme is sub-divided (grouped) into 4 different sub-transformation schemes, the resulting sub-IDs would be *"ID 3-1"*, *"ID 3-2"*, *"ID 3-3"* and *"ID 3-4"*. These IDs are also passed among computing nodes and headnode to identify a transformation task, to proceed with further processing or to recall a process after a computing node failure. Internally each transformation task can be further subdivided, which mostly depends on the dynamic applicability of cited FermaT transformations and their corresponding AST tree path.

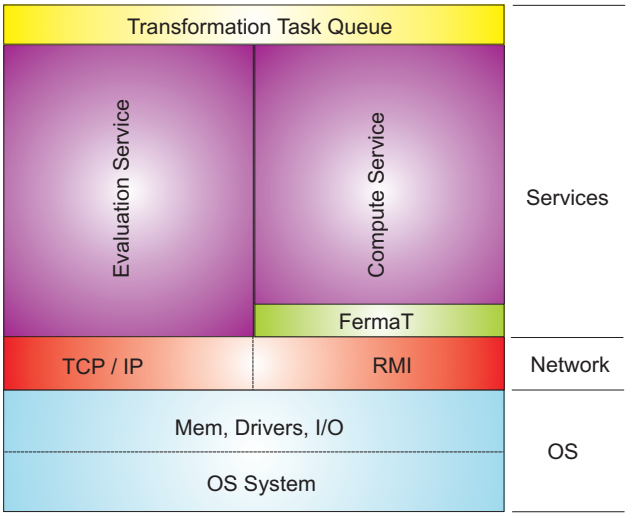


FIGURE 5.7: Computing Node Services

5.3.1 The Evaluation Service

The evaluation service evaluates the transformation processing results of each transformation applied within a transformation task. As already demonstrated, once a FermaT transformation has been applied on a specific AST node within a WSL program source, the resulting program "P_{i+1}" has to be evaluated according to its task specification. As tasks can be further refined by constraints, they need to be evaluated. Constraints can be a type of parallel processing constraint trying to fulfil a particular processing result, such as speed-up, or specifiy that the resulting WSL programs should fulfil a reengineering constraint. During the processing of a transformation task, usually a couple of hundred or even thousands of transformations need to be applied. Since those transformations are normally expressed via transformation sequences, they are computed in a special manner. This process is further specified in Chapter 3.4.

5.3.2 The Communication Service

The communication service established for computing nodes has the same functions as the one specified for the headnode. As a result, the same TCP/IP aand RPC network communication modules are utilised within the computing nodes. The alternation between the two can be performed by switching the headnodes communication procedure from TCP/IP to RPC communication or vice versa. Alternatively this can be achieved through parallel transformation task description refinement by utilising the developed parallel transformation processing language.

5.4 Summary

This chapter gave an overview of the services which have been developed for the proposed parallel transformations framework. The services are categorised between headnode and computing node services. The chapters focus has been on system features of the headnode, as they are designed to lead to a successful fulfilment of the reengineering aim. The developed headnodes analysing systems tries to support and fulfil this direction. In addition for a successful computation, computing node processing steps and services are also reviewed.

Chapter 6

Laws of Decomposition

Objectives

- To describe the purpose of decomposing transformation scheme descriptions.
- To present decomposition laws.
- To provide proof through examples.

6.1 Introduction

This chapter introduces and explains the laws defined to decompose transformation scheme descriptions for parallelisation purposes. This parallelisation is necessary for two reasons. The main one is to accelerate the computation of transformation tasks. Secondly, the illustrated process should decrease the generated transformation sequence search space by subdivision and therefore eliminate redundant transformation sequences.

As an example, the transformation scheme description utilised in case study 2 (Chapter 9.7) produces 37.400 transformation sequences. Taking into account that each transformation sequence has an average length of 12 transformations and it takes 6 seconds to compute one transformation sequence, with an average transformation computation time of 0.5 second, the overall computation time would result in over 62 hours.

To remedy and abbreviate this situation, laws to decompose transformation scheme descriptions are proposed. These laws allow transformation scheme descriptions to subdivide and filter. The results are schemes which can be computed more efficiently in parallel than within a single system. The parallel transformations framework presented within this thesis assists within this process.

6.2 Laws for Transformation Scheme Description Decomposition

The following sections outline and explain the definition and usage of laws to decompose transformation scheme descriptions. These laws can be utilised to decompose transformation schemes into smaller sub-schemes. The resulting sub-schemes are independent constructs and can be computed in parallel. The combination of all sub-schemes would lead to the same original transformation scheme description construct. Before speaking of the term law, what exactly is meant by the expression "*law*" needs to be classified.

The specified laws classify and extract the behaviour and order of program transformations. These laws simply characterise the semantics of the underpinning transformation scheme description language. These laws arise from informal understanding on how transformation scheme descriptions are constructed and how they are applied. These laws allow a precise description of their operations within the program transformation domain to be given. The laws given are congruences in the term of denotational semantics of transformation scheme descriptions. The laws quoted must be true in any reasonable abstract transformation scheme description, its semantics and transformation scheme.

All laws have the same form of "**P** = **Q**", whereas "**P**" and "**Q**" both represent processes. Informally this must mean that "**P**" is essentially the same as "**Q**". To an observer who cannot detect their internal structure, the behaviour of "**P**" and "**Q**" are indistinguishable. Since the laws are used to transform compound transformation scheme descriptions into subcomponents, "**P** = **Q**" must be true, if a transformation scheme description should be transformed into its corresponding sub-schemes. Thus "**P** = **Q**" does not mean that "**P**" and "**Q**" run at the same speed, neither that they require the same amount of storage within the presented environment.

6.3 Lexical Units for Decomposition

To find an efficient way to compute transformation scheme descriptions in parallel, schemes defined by the maintainer need to be analysed by the headnode. In order to do so, lexical units need to be identified. These units are used to degrate the outlined transformation scheme descriptions into its constructing parts. The resulting parts can be computed in parallel. As transformation scheme descriptions are embedded within parallel transformation tasks, it needs to be distinguished between lexical component **keywords** and **symbols** of both definitions. Based on their syntax and specification described in Chapter 3.3 and Chapter 4.6 the following lexical units (*keywords* and *symbols*) may occur within parallel transformation task definitions:

Keywords: All keywords are upper case (e.g. "*PAR*"), possibly with digits (e.g. "T_1"). All keywords are reserved and thus may not be used by the maintainer for other purposes than specified below:

- **"PAR":** The introduction of the "*PAR*" keyword expresses parallelisation, similar to the symbol "‖" defined below. The difference is the keyword "*PAR*" directly introduces possible parallelisation within the definition of a transformation task. Whereas "‖" is used within the transformation scheme description analysis process to mark and identify the parallelisation of transformation processes.
- **"T_i":** States a transformation within a transformation scheme description. In this context, it can be any transformation of the FermaT transformation catalogue.
- **"C_n":** Expresses and defines a constraint specified and explained in Chapter 3.4.

Symbols: Symbols are utilised to separate, characterise transformation processes as the syntax of transformation scheme descriptions specifies:

- **",":** Introduces a sequence of transformations.
- **"|":** Declares an alternative construct of transformations.
- **"[ ]":** Specifies a quantifier construct, exp. 1..3.
- **"{}":** Represents the "0" iteration of a decomposed quantifier construct. It can be interpreted as a "**skip**" within the transformation process, meant to follow the next transformation processing step within a sequence or alternative operation.
- **"‖":** Expresses possible parallel computation of transformations or transformation sequences.

In general, transformation sequences are generated by transformation schemes. These schemas are created via automaton construction. Transformation scheme descriptions outline and characterise schemes and are primarily expressed via a formal language. To run the presented technique efficiently in parallel, the original transformation schemes descriptions need to be decomposed before the transformation schema is constructed and executed.

6.4 Laws of Decomposition

According to the specification above and the formal language to describe and outline transformation scheme descriptions, the following laws characterise transformation processes:

(1) $T_1 = T_1$
(2) $T_1, T_2 = T_1, T_2$
(3) $T_1 \mid T_1 = T_1$
(4) $T_1 \mid T_2 = T_2 \mid T_1$
(5) $T_1\ [0..3] = (\{\})\mid (T_1) \mid (T_1, T_1) \mid (T_1, T_1, T_1)$

With the following specification:

- **Law (1):** If the same transformation "T_i" of a transformation catalogue should produce the same program state "P_j", the same program transformation "T_i" has to be applied on the same program state.
- **Law (2):** If a program state result of a transformation sequence ("P_i, P_j") should result in an equivalent state, the same transformation sequence need to be applied on the same program code in the same order.
- **Law (3):** If equally specified program transformations are stated within an alternative construct, the term can be simplified to the usage of the single transformation "T_1".
- **Law (4):** The application order of transformations within an alternative construct ("$T_i..T_n$") on a particular Program state "P_i" is not crucial. Causing all transformations within the same alternative construct specification and same program state "P_j" can be reversed or changed.

- **Law (5):** Represents an "alternative iteration". The given example demonstrates this alternative way with transformation "T_1", resulting in 4 transformation sequences (0 through 3: "({}) | (T_1) | (T_1, T_1) | (T_1, T_1, T_1))").

6.5 Laws of Parallelisation

To denote parallelisation possibilities within a transformation scheme description, this approach introduces the "*PAR*" keyword. By utilising this specification on the rules above, the corresponding "*PAR*" constructs are:

(6) PAR (T_1) = T_1
(7) PAR (T_1, T_2) = T_1, T_2
(8) PAR (T_1 | T_1) = T_1
(9) PAR (T_1 | T_2) = T_1 | T_2
(10) PAR (T_1 | T_2) = PAR (T_2 | T_1)

- **Law (6):** Expresses, when a single transformation "T_1" can be performed in parallel it can be also performed sequentially.
- **Law (7):** States, when a sequence of transformations as ("T_1, T_2") can be computed in parallel, this process can be also performed in a sequential manner.
- **Law (8):** To compute transformation "T_1" and "T_1" in parallel, the same program state result can be achieved by performing the single transformation "T_1" sequential.
- **Law (9):** If two transformations "T_1" and "T_2" can be performed independently in parallel, they can be also performed independent sequentially.
- **Law (10):** Utilising law (9), it does not matter in which order the two transformations "T_1" and "T_2" are performed in parallel, the resulting program states will not differ.

6.6 Laws of Associations

On the basis of these laws and its transformation scheme description constructs, very simple associative laws can be specified, similar to associative laws in the binary sense of "**a * (b * c) = (a * b) * c**". In analogous to this and in context of transformation scheme description construction, it can take an arbitrary finite number of arguments.

(11) PAR $(T_0, (T_1 \mid T_2)) = \text{PAR } (T_0, T_1) \parallel \text{PAR } (T_0, T_2)$

(12) PAR $((T_1 \mid T_2), (T_1 \mid T_2)) = (T_1 \mid T_2), (T_1 \mid T_2)$
$= (T_1, T_1) \parallel (T_1, T_2) \parallel (T_2, T_1) \parallel (T_2, T_2)$
$= (T_1 \mid T_2), \text{PAR } (T_1 \mid T_2)$

(13) PAR $((T_1 \mid T_2), (T_1 \mid T_2)) = ((T_1 \mid T_2), T_1) \parallel ((T_1 \mid T_2), T_2)$
$= (T_1, T_1) \parallel (T_2, T_1) \parallel (T_1, T_2) \parallel (T_2, T_2)$
$= (T_1 \mid T_2), (T_1 \mid T_2)$

- **Law (11):** States that a transformation sequence construct which consists of one transformation ("T_0") and an alternative-construct ("$T_1 \mid T_2$"), can be decomposed into two independent constructs ("T_0, T_1") and ("T_0, T_2") and computed in parallel.
- **Law (12):** A sequence of alternative constructs can be decomposed for parallelisation, by combining the transformations of each alternative-construct. This process would start with the combination of the first transformation of the first alternative-construct with the first transformation of the second alternative-construct. The result will be four independent transformation sequences. These constructs can be computed in parallel or in a sequential manner according to law (6) or (9).
- **Law (13):** Another option compared to law (11) is the decomposition of an alternative-construct sequence into two independent constructs, resulting in more independent sequences of alternative constructs.

As illustrated by these laws, parallelisation of transformation scheme descriptions can be achieved by applying the above stated laws to transformation sequence- and transformation alternative-constructs. The following sections demonstrate the applicability of these laws on very commonly used iteration and alternative constructs of transformation scheme descriptions.

6.7 Decomposition Algorithm

The following demonstrates how the transformation scheme description decomposition algorithm operates based on the laws specified above. Simple iteration- and alternative-constructs examples are given. As explained earlier, this algorithm technique is one of the key features used for automatic parallelisation of transformation processes. Any description submitted for parallel computation is decomposed in the following manner:

1. Evaluation of the number of quantifier constructs and their interval ranges within the specified transformation scheme description. The weighted numbers are used for further calculation purposes such as parallelisation possibilities and speed-up prediction techniques.

2. As a pre-processing step within the decomposition algorithm, each quantifier construct ("[n ... m]") is substituted with a sequence of transformations or an alternative-construct of transformations as specified within law (5).

3. Each substitution construct is decomposed according to its definition and laws illustrated above.

4. This results in independent transformation scheme description constructs which can be conducted in a parallel manner.

5. Two intentions are followed by this technique:

   - To produce more alternatives of transformation scheme constructs. This has been demonstrated by the described algorithm and the laws above.
   - The second reason and main reason for this technique is to further speed-up the transformation tasks. This technique is further explained in Section 6.12.

Within the following, a short transformation scheme description parallelisation example is given. The general procedure performs in the following way: At first, the algorithm reads the transformation scheme description defined by the formal language. Since it is easier to handle short transformation names, the original FermaT transformation names are substituted by short substitutes illustrated below.

(1) (T_0, T_1 [0..4], T_2, (T_3 [0..3] | T_4)) {C_1}

The next processing step includes the applicability of the specified laws on iteration and alternative constructs. In the case of the first iteration and quantifier construct ("T_1 [0..4]") ranging from 0 to 4, five alternative ways can be achieved and generated by the application of law (5). This results in:

(0) ({}) |
(1) (T_1) |
(2) (T_1, T_1) |
(3) (T_1, T_1, T_1) |
(4) (T_1, T_1, T_1, T_1)

Inserting the generated constructs into the original transformation scheme expression, results in five independent transformation sub-schemes listed below.

(0) $(T_0, (\{\}), T_2, (T_3\ [0..3] \mid T_4))\ \{C_1\}\ \|$
(1) $(T_0, (T_1), T_2, (T_3\ [0..3] \mid T_4))\ \{C_1\}\ \|$
(2) $(T_0, (T_1, T_1), T_2, (T_3\ [0..3] \mid T_4))\ \{C_1\}\ \|$
(3) $(T_0, (T_1, T_1, T_1), T_2, (T_3\ [0..3] \mid T_4))\ \{C_1\}\ \|$
(4) $(T_0, (T_1, T_1, T_1, T_1), T_2, (T_3\ [0..3] \mid T_4))\ \{C_1\}$

These transformation sub-schemes can be computed in parallel.

6.8 Laws of Quantifier Construct

The quantifier construct is a commonly used technique to express the repeated application of a transformation on a specific program state "P_j". Listing 6.1 illustrates a simple example in which the transformation "*Simplify If*" should be applied between 1 and 4 times on a particular WSL program code section. At this point the focus lies on the scheme expression and its parallelisation possibilities and therefore leaves the specification of the constraint "C_1" aside.

```
1 (
2   < Simplify If @ T_Cond > [1 .. 4]
3 ) {C1}
```

Listing 6.1: Quantifier Construct

To simplify the presented transformation scheme description the table below is used to substitute its transformations.

Transformations	Acronym
< Simplify If @ T_Cond >	T_0

Table 6.1: Quantifier Construct Substitution

Introducing an imaginary "*PAR*" keyword to the transformation scheme description above plus the application of the decomposition law (5) to the quantifier construct, would result in four transformation sub-schemes which can be computed in parallel.

(1) $(T_0)\ \{C_1\}\ \|$
(2) $(T_0, T_0)\ \{C_1\}\ \|$

(3) $(T_0, T_0, T_0)\ \{C_1\}\ \|$
(4) $(T_0, T_0, T_0, T_0)\ \{C_1\}$

6.9 Laws of Alternative Constructs

The decomposition of an "*alternative construct*" is another possibility to achieve parallelisation. In general it has to be distinguished between three different types of alternative constructs which are explained within the following sub-sections.

- *Alternative construct* with no quantifier construct (Type I);
- *Alternative construct* with an inside quantifier construct (Type II);
- *Alternative construct* with outside quantifier construct (Type III);

Listing 6.2 gives a simple example of an "*alternative construct Type I.*" The transformation scheme description embeds two transformations: "*Simplify If*" and "*Simplify*".

```
1 (
2   < Simplify If @ T_Cond > |
3   < Simplify @ // >
4 ) {C1}
```

Listing 6.2: Alternative Construct Type I

To simplify the presented transformation scheme description the table below is used to substitute its transformations.

Transformations	Acronym
< Simplify If @ T_Cond >	T_0
< Simplify @ // >	T_1

Table 6.2: Alternative Construct Type I Substitution

$T_0 \mid T_1 = \text{PAR}\ (T_0 \mid T_1)$

And states, if there exists a transformation construct within a transformation scheme description in which transformations are grouped as an alternative, they can be considered as independent and can be executed in parallel (law (8)). Listing 6.3 presents a more complex "*alternative construct*".

```
(
  < Simplify Action System @ // >,
  < Delete All Redundant @ // >,
  < Abort Processing @ // >,
  (
    < Simplify If @ T_Cond > |
    < Simplify @ // >
  ) {C1}
) {C2}
```

LISTING 6.3: Alternative Construct Decomposition

Transformations	Acronym
< Simplify Action System @ // >	T_0
< Delete All Redundant @ // >	T_1
< Abort Processing @ // >	T_2
< Simplify If @ T_Cond >	T_3
< Simplify @ // >	T_4

TABLE 6.3: Alternative Construct Substitution

With the given substitution table, the above stated transformation scheme description can be expressed as:

$$(T_0, T_1, T_2, (T_3 \mid T_4)\{C_1\}\ )\{C_2\}$$

If law (10) is applied to decompose the internal sub-scheme ("$(T_3 \mid T_4)$"), the outcome would be two transformation sub-schemes (1) and (2).

(1) $(T_0, T_1, T_2, (T_3)\{C_1\}\ )\{C_2\}$ ‖
(2) $(T_0, T_1, T_2, (T_4)\{C_1\}\ )\{C_2\}$

6.9.1 Laws of Alternative Construct Type II

The second type of an "*alternative-construct*" is one in which the quantifier construct is stated inside the sub-scheme alternative-construct. Listing 6.4 gives a demonstration of such an example in which the transformation *"Simplify If"* should be applied between "1" and "4" times on a particular WSL program state "P_j". Table 6.4 specifies the transformation substitution table of this listing.

```
(
    < Simplify If @ T_Cond > [1..4] |
    < Simplify @ // >
){C1}
```

Listing 6.4: Alternative Construct Type II

Transformations	Acronym
< Simplify If @ T_Cond >	T_0
< Simplify @ // >	T_1

Table 6.4: Alternative Construct Type II Substitution

Utilising the above table and the application of law (5), the transformation scheme description expressed could be also specified to the form of:

$$((T_0 \mid (T_0, T_0) \mid (T_0, T_0, T_0) \mid (T_0, T_0, T_0, T_0)) \mid T_1)\ \{C_1\}$$

The combination of law (6) law (7) would eliminate the "alternative ("|") construct", the result are 4 transformation sub-schemes alternative constructs.

(1) $(T_0 \mid T_1)\ \{C_1\}$
(2) $((T_0, T_0) \mid T_1)\ \{C_1\}$
(3) $((T_0, T_0, T_0) \mid T_1)\ \{C_1\}$
(4) $((T_0, T_0, T_0, T_0) \mid T_1)\ \{C_1\}$

Further elimination of the "|" construct and transformation "T_1", would generate the following constructs. Additionally applying law (8) for a separate parallel computation step, would result in five independent transformation sub-schemes which can be computed in parallel.

(1) $(T_1)\ \{C_1\} \parallel$
(2) $(T_0)\ \{C_1 \parallel$
(3) $(T_0, T_0)\ \{C_1\} \parallel$
(4) $(T_0, T_0, T_0)\ \{C_1\} \parallel$
(5) $(T_0, T_0, T_0, T_0)\ \{C_1\} \parallel$

6.9.2 Laws of Alternative Construct Type III

As defined, the "*alternative construct Type III*" can be identified by the usage of a stated quantifier construct at the end of a transformation scheme or sub-scheme description. Its construct is marked red in Listing 6.5.

```
1 (
2  (
3   < Simplify If @ T_Cond > |
4   < Simplify @ // >
5  )[1..3]
6 ){C1}
```

Listing 6.5: Alternative Construct Type III

To simplify the presented transformation scheme description the table below is used to substitute its transformations.

Transformations	Acronym
< Simplify If @ T_Cond >	T_0
< Simplify @ // >	T_1

Table 6.5: Alternative Construct Type III Substitution

According to the definition of the alternative-construct defined and the usage of law (5) for the decomposition of quantifier constructs, the transformation scheme description can be decomposed for each alternative iteration.

(1) $(T_0 \mid T_1)\ \{C_1\}$
(2) $(T_0 \mid T_1), (T_0 \mid T_1)\ \{C_1\}$
(3) $(T_0 \mid T_1), (T_0 \mid T_1), (T_0 \mid T_1)\ \{C_1\}$

Utilising law (11) to decompose the above sequence transformation constructs (2) and (3), the resulting transformation sub-schemes are:

(1) $T_0 \parallel T_1$
(2) $(T_0, T_0) \parallel (T_0, T_1) \parallel (T_1, T_0) \parallel (T_1, T_1)$
(3) $(T_0, T_0, T_0) \parallel (T_1, T_0, T_0) \parallel (T_0, T_1, T_0) \parallel (T_0, T_0, T_1) \parallel (T_1, T_1, T_0) \parallel$
$(T_1, T_0, T_1) \parallel (T_0, T_1, T_1) \parallel (T_1, T_1, T_1)$

It has to be stated that the citation of constraint "C_1" has been left aside, because the focus has been on the decomposition of transformation scheme descriptions.

6.10 Laws of Combination of a Qualifier and Alternative Construct

The following example illustrates the combination of a quantifier as well as an alternative construct within a transformation scheme description. It demonstrates how the decomposition algorithm performs to decompose a scheme for parallelisation purposes. Listing 6.6 presents an example of such a construct. The highlighted quantifier construct (red) states where possible parallelisation can be achieved.

The first possibility to realise parallelisation can be achieved by the decomposition of the transformation "*Use Assertion*" and its quantified construct, followed by the second opportunity, the decomposition of the alternative sub-scheme, starting with transformation "*Simplify If*".

```
1 (
2   < Simplify Action System @ // >,
3   < Use Assertion @ T_Assert > [0 .. 4],
4   < Abort Processing @ // >,
5   (
6     < Simplify If @ T_Cond >  [0 .. 3] |
7     < Simplify @ // >
8   ) {C1}
9 ) {C2}
```

Listing 6.6: Alternative And Quantifier Transformation Scheme Description

The Table 6.6 gives the substitution of the transformations used.

Transformations	Acronym
< Simplify Action System @ // >	T_0
< Use Assertion @ T_Assert >	T_1
< Abort Processing @ // >	T_2
< Simplify If @ T_Cond >	T_3
< Simplify @ // >	T_4

Table 6.6: Alternative And Quantifier Transformation Scheme Description Substitution

Given the transformation scheme description above, the scheme is substituted by the acronyms stated resulting in the following specification: **(1)**: "(T_0, T_1 **[0..4]**, T_2, (T_3 **[0..3]** | T_4){C_1}){C_2}"

The detected quantifier constructs within the scheme are:
(2): T_1 [0..4]
(3): T_3 [0..3]

According to the definition of law (5) and the formal language specification, the constructs (1) and (2) can be decomposed to the following:

(4): ({} | (T_1) | (T_1, T_1) | (T_1, T_1, T_1) | (T_1, T_1, T_1, T_1))
(5): ({} | (T_3) | (T_3, T_3) | (T_3, T_3, T_3))

Taking transformation scheme description (1) and substituting its constructs "T_1 [0..4]" and "T_3 [0..3]" with transformation sub-schemes (4) and (5), the following constructs are generated:

(6): (T_0, T_1 **[0..4]**, T_2, (T_3 **[0..3]** | T_4){C_1 }{C_2}
= (T_0, ({} | (T_1) | (T_1, T_1) | (T_1, T_1, T_1) | (T_1, T_1, T_1, T_1)), T_2, (T_3 **[0..3]** | T_4){C_1 }{C_2}

= (T_0, ({} | (T_1) | (T_1, T_1) | (T_1, T_1, T_1) | (T_1, T_1, T_1, T_1)), T_2, (({} | (T_3) | (T_3, T_3) | (T_3, T_3, T_3)) | T_4){C_1} }{C_2}

To decompose the whole construct to achieve independent transformation sequences, the algorithm starts at the far left with the first substitution (4), takes the first transformation alternative, this would be the empty set "{}" and combines it will all other possibilities of the second substitution (5), which results in the following transformation sequences:

(T_0, {}, T_2, {}) ‖
(T_0, {}, T_2, T_3) ‖
(T_0, {}, T_2, T_3, T_3) ‖
(T_0, {}, T_2, T_3, T_3, T_3) ‖
(T_0, {}, T_2, T_4)

For the second transformation which is ("T_1") and the combination of all possibilities of the second substitution (5), this would result in the following sequences:

(T_0, T_1, T_2, {}) ‖
(T_0, T_1, T_2, T_3) ‖
(T_0, T_1, T_2, T_3, T_3) ‖
(T_0, T_1, T_2, T_3, T_3, T_3) ‖
(T_0, T_1, T_2, T_4)

For the second transformation which is ("T_1") and the combination of all possibilities of the second substitution (5), the results would be:

$(T_0, T_1, T_1, T_2, \{\}) \parallel$
$(T_0, T_1, T_1, T_2, T_3) \parallel$
$(T_0, T_1, T_1, T_2, T_3, T_3) \parallel$
$(T_0, T_1, T_1, T_2, T_3, T_3, T_3) \parallel$
$(T_0, T_1, T_1, T_2, T_4)$

$(T_0, T_1, T_1, T_1, T_2, \{\}) \parallel$
$(T_0, T_1, T_1, T_1, T_2, T_3) \parallel$
$(T_0, T_1, T_1, T_1, T_2, T_3, T_3) \parallel$
$(T_0, T_1, T_1, T_1, T_2, T_3, T_3, T_3) \parallel$
$(T_0, T_1, T_1, T_1, T_2, T_4)$

$(T_0, T_1, T_1, T_1, T_1, T_2, \{\}) \parallel$
$(T_0, T_1, T_1, T_1, T_1, T_2, T_3) \parallel$
$(T_0, T_1, T_1, T_1, T_1, T_2, T_3, T_3) \parallel$
$(T_0, T_1, T_1, T_1, T_1, T_2, T_3, T_3, T_3) \parallel$
$(T_0, T_1, T_1, T_1, T_1, T_2, T_4)$

The resulting decomposition constructs are 25 independent transformation sequences. They all could be computed in parallel. This unrolling procedure is one of the key features performed by the headnode. It only consumes a minimum of the overall processing time, compared to the appliance of the 135 transformations provided in this example. The estimated overall processing time would be between 1 - 1.5 minutes. To avoid redundant work the next section introduces a technique in which redundant transformation sequences are eliminated and independent sequences are grouped for computing node assignment.

6.11 Eliminating Redundant Transformation Sequences

One of the key aspects of this decomposition and special grouping function is, to eliminate redundant work during parallel processing. This can be avoided by eliminating the generated transformation sequences which are already included within the transformation sequence search space. To give an example the following sequences "1 - 4" can be combined into one grouped sequence ("1-4"). Nonetheless it needs to be ensured that the stated reengineering constraint "C_1" is checked for fulfilment after each transformation process, within each WSL program "P_i" program state for possible satisfaction.

(1) (T_0) $\{C_1\}$
(2) (T_0, T_0) $\{C_1\}$
(3) $(T_0, T_0, T_0)\{C_1\}$

(4) (T_0, T_0, T_0, T_0) $\{C_1\}$

(1-4) ((T_0, T_0, T_0, T_0) $\{C_1\}$

Within the process of program transformation application, the resulting WSL program "P_{i+1}" and its corresponding AST have to be loaded into the FermaT transformation. Since this process has to be performed after each transforming step, the fulfilment of constraint "C_1" can be validated during each process. To avoid redundant work during transformation processing within each computing node, the assigned transformation sub-schemes and the resulting transformation sequences are computed under the presented technique. Further computation speed-up can be achieved by the evaluation if the current WSL program "P_i" has already been computed. If so, the specific "P_i" program state can be recalled and the transformation processing can be continued from that program state on.

6.12 Grouping Transformation Sequences

The example in Section 6.10 illustrated that only 5 transformations in combination with quantifier- and alternative-constructs can produce up to 25 different sequences and 135 transformation processing circles. More complex examples which are discussed in Chapter 9 within the case studies can produce up to 37400 individual transformation sequences. The result will be the application of nearly 77626 transformations, resulting in an overall computing time of over 21 hours on a single processing PC. To decrease this tremendous amount of transformation sequences not only parallelising techniques are introduced, also a transformation sequence grouping algorithm has been specified. There are two reasons for introducing this kind of technique. The first one is to be able to assign more efficiently transformation sequences to the computing nodes. On the other hand and this is probably the most important purpose, is to reduce redundant computing work. The result of this is that the computing nodes compute more efficiently while at the same time communication overhead is reduced. The following gives a short example of the grouping technique. As an example, a transformation scheme description produces the transformation sequences stated below.

(1) (T_0, {}, (T_2)$\{C_1\}$, {})$\{C_2\}$ ‖
(2) (T_0, {}, (T_2)$\{C_1\}$, T_3)$\{C_2\}$ ‖
(3) (T_0, {}, (T_2)$\{C_1\}$, T_3, T_3)$\{C_2\}$ ‖
(4) (T_0, {}, (T_2)$\{C_1\}$, T_3, T_3, T_3)$\{C_2\}$ ‖
(5) (T_0, {}, (T_2)$\{C_1\}$, T_4)$\{C_2\}$

(6) (T_0, T_1, (T_2){C_1}, {}){C_2} ‖
(7) (T_0, T_1, (T_2){C_1}, T_3){C_2} ‖
(8) (T_0, T_1, (T_2){C_1}, T_3, T_3){C_2} ‖
(9) (T_0, T_1, (T_2){C_1}, T_3, T_3, T_3){C_2} ‖
(10) (T_0, T_1, (T_2){C_1}, T_4){C_2}

The algorithm would operate as follows:

It reads the first two transformations of each transformation sequence and compares each transformation. "{}" in this example is symbolising a skip in the transformation process and has been left inside each transformation sequence after the unrolling procedure, to easier evaluate the difference of transformation sequences. In this example, the second transformation already differs within all transformation schemes. The algorithm starts separating the 10 sequences into two groups of 5 sequences each as shown above. Next, the algorithm evaluates each group individually. During this splitting process, the algorithm also creates for each group an overall group transformation description which ends up representing the whole sub-group of transformation sequences as a single expression. In the case of the first group, this sequence-group would already include the first two expressions ("T_0, {}"). Since the third transformation and its constraints are identical for all transformation sequences, they are also added to the group scheme as additional transformations. Recognising that all sequences within this group differ at transformation state 4, an alternative-construct is created in which all possibilities are cited as separate alternatives. At the same time the algorithm also notices that transformation sequence (2), (3) and (4) do not differ, in this case it is likely that more of the same transformation occur within those individual sequences, which results in a "(T_3 | (T_3, T_3) | (T_3, T_3, T_3))" alternative. Recognising that the end of the transformation sequence is reached by discovering the closing-bracket, the last constraint "{C_2}" is read and added to the sub-group scheme.

Performing this step for both groups, the results are:

Group (1): (T_0, {}, T_2, ({} | T_3 | (T_3, T_3) | (T_3, T_3, T_3) | T_4){C_1}){C_2}
Group (2): (T_0, T_1, T_2, ({} | T_3 | (T_3, T_3) | (T_3, T_3, T_3) | T_4){C_1}){C_2}

A positive side-effect of this technique is that the resulting groups are easier to divide for parallel computation. With the slightly different algorithm stated above those constructs can even be subdivided into smaller groups and assigned to more computing nodes. These parallel processing techniques are discussed in Chapter 7. To speed-up the parallel computation and run it more economically, the following alternative decomposition example is provided. The embedded alternative-construct serves as an ideal way to achieve parallelism. The sample below includes "4 transformations" of which all

of them could be computed between "1 - 3" times on the same WSL program state "P_i", expressed via a quantifier construct.

$((T_0 \mid T_1 \mid T_2 \mid T_3)[1 \,..\, 3]\,)\{C_1, C_2\}$

According to law (5) this construct could be decomposed and expressed as:

$((T_0 \mid T_1 \mid T_2 \mid T_3),(T_0 \mid T_1 \mid T_2 \mid T_3)[0 \,..\, 2]\,)\{C_1, C_2\}$

To achieve parallelism the above stated constructs can be decomposed by utilising law (13). The only limitation this decomposition procedure has is the quantifier construct, limiting the total number of transformation application to three. Assuming that this expression needs to be parallelised only for two computing nodes, the decomposition algorithm would unroll the transformation scheme to the following two constructs:

$((T_0 \mid T_1),(T_0 \mid T_1 \mid T_2 \mid T_3)[0 \,..\, 2]\,)\{C_1, C_2\}$
$((T_2 \mid T_3),(T_0 \mid T_1 \mid T_2 \mid T_3)[0 \,..\, 2]\,)\{C_1, C_2\}$

To further speed-up the overall computation task these transformation sub-schemes could be decomposed for the submission to 4 computing nodes, resulting in:

$((T_0),(T_0 \mid T_1 \mid T_2 \mid T_3)[0 \,..\, 2]\,)\{C_1, C_2\}$
$((T_1),(T_0 \mid T_1 \mid T_2 \mid T_3)[0 \,..\, 2]\,)\{C_1, C_2\}$
$((T_2),(T_0 \mid T_1 \mid T_2 \mid T_3)[0 \,..\, 2]\,)\{C_1, C_2\}$
$((T_3),(T_0 \mid T_1 \mid T_2 \mid T_3)[0 \,..\, 2]\,)\{C_1, C_2\}$

By utilising these procedures and laws as a foundation, the headnode's analysing system is able to discover transformation schemes which are not applicable. This results in the avoidance of redundant work by eliminating unsatisfactory or not applicable transformation sequences. Utilising this technique is also a reason why the case studies examples presented are computed more efficiently.

6.13 Summary

This chapter described the fundamental parts for the decomposition of transformation scheme descriptions. It illustrated the laws defined and proofs of correctness of decomposition. It demonstrated its law procedures on common quantifier- and alternative-constructs and explained algorithms for decomposition transformation scheme descriptions. It presented transformation sequence grouping techniques to group and assign generated sub-schemes more economically to computing nodes, while transformation sequence elimination techniques further assist the parallelisation process, by eliminating inapplicable transformation sequences.

Chapter 7

Parallel Transformation Processing and Task Distribution

Objectives

- Presentation of the applicability of a transformation sequence.
- Description and outline of parallel transformations processes.
- Map parallel transformations processes to a parallel environment.
- Success and constraint satisfaction of parallel transformations tasks.

7.1 Introduction

This chapter describes the parallel transformations processing techniques proposed within this thesis. It outlines and demonstrates how parallel transformations tasks and transformation scheme descriptions can be mapped to a parallel computing environment. Today, parallel processing is mainly used to speed-up computation tasks. Parallel techniques have not been widely used within the reengineering domain. The following sections illustrate how parallel computation can be utilised within this field and how it has been approached.

7.2 Transformations and Transformation Sequences

To restructure WSL program code within a reengineering task, the FermaT transformation engine and its transformation catalogue are irreplaceable. FermaT's reengineering features are mainly focused on International Business Machines (IBM) 360 assembler code. Utilised transformations are mathematical proven and only change the program structure while its semantics are preserved.

During a FermaT reengineering process, transformed WSL programs serve as intermediate processing steps. WSL programs are within the transformation engine represented as lists and inner lists. To traverse through those lists special commands are needed. In order to apply a transformation within this list, it is not only mandatory to test if the transformation is applicable, it also needs to be evaluated on which Abstract Syntax Tree (AST) path the transformation could be applied.

7.2.1 The Abstract Syntax Tree (AST) Path

In most cases the evaluation of the AST path is mandatory to apply a transformation. It has to be distinguished between transformations which are only applicable on a specific program AST path and transformations which are applicable on any AST tree node. The full list of AST paths and FermaT specific transformations is outlined in Appendix A. Within a reengineering context, the more precise the AST path and its corresponding transformation are specified, the more precise the outcome will be. Usually system maintainers are unfamiliar with all transformations, they are unsure on which AST path a transformation should be applied or he/she is doubtful which effects a transformation will have on the program code. To ease this situation, the maintainer is given a language to outline and describe transformation processes, by the utilisation of the presented transformation scheme description language. In the context of transformation application and AST path citation the following is specified:

> "<FermaT Transformation> @ <AST path >"

where <**AST path** > is either:

- **A Definite AST Path** is expressed via a number of indices. Each AST node within a WSL program is automatically indexed during the analysing process. In this context, the definite AST path index is specified after the citation of a WSL transformation and the "@" symbol. Listing 7.2 presents an example of such.

```
1 < Simplify @ // >
```

Listing 7.1: Definite AST Path

In this example the transformation "*Simplify*" should be applied on the AST node with the index "//". This index symbolises an empty set and specifies the root node ("//") of a WSL program. To transfer this to a concrete example, Figure 7.1 gives an illustration. The root node("//") can be found at the top. The next level with the WSL program would be the states "/0,0/", "/0,1/", "/0,2/" and "/0,3/" and are considered as sub-paths of "/0/". The indexing process is part of the analysing system described in Chapter 5.2.2. Within Figure 7.2 all black values are indices of program AST types.

- **A Restricted AST Path** is specified similar to a definite AST path whereas the last index is either an asterisk, a plus or a question mark. The difference is:
  - **The Asterisk** indicates that all AST sub-types and the AST node itself are taken into consideration for a transformation process. For instance, the restricted AST path "/0,*/" takes the AST node at index "/0/" and all sub-node types into consideration. In Figure 7.1 this process would include all AST nodes within the red marked boundary.
  - **The Plus** indicates that only the subtypes are considered. The restricted AST path "/0,+/" selects only the subtypes of the definite AST path "/0/". In Figure 7.1 this would include the AST nodes "/0,0/", "/0,1/", "/0,2/" and "/0,3/" and its corresponding branches.
  - **The Question Mark** Mark indicates that only the direct subtypes of the selected AST type are selected. The restricted AST path "/0,?/" selects only the direct subtypes of the AST type at the definite AST path "/0/". In Figure 7.1 this would be the AST nodes "/0,0/", "/0,1/", "/0,2/" and "/0,3/". Simply all AST nodes within the blue marked boundary.
- **An Unrestricted AST Path** is expressed with an asterisk after a transformation. The path could be either specified as "$< T_i$ @ /*/ $>$" or by a transformation alone "$< T_i >$". Not stating any AST path leads alternatively to a massive consumption of the appliance of the given transformation.
- **An Abstract AST Path** is stated on the basis of a specific AST type. An example of how an abstract path is cited is illustrated in 7.2. By this abstract definition, the complete WSL program will be searched for the specified AST type. Within the given example, the transformation "*Simplify IF*" would be applied on all "*T_Cond*" AST nodes.

```
1 < Simplify If @ T_Cond >
```

Listing 7.2: Abstract AST Path

- **The Abstract Restricted AST Path**

  is introduced to avoid any massive search. In the case of Listing 7.2, the abstract restricted search path of the specific AST type "*T_Cond*" can be specified with "/,+/", illustrated in Listing 7.3. The search space for this specification would include all sub-types of the root node ("//") and would result in the search space of "/0/".

```
1 < Simplify If @ T_Cond: /,+/>
```

Listing 7.3: Abstract Restricted AST Path

To illustrate the complexity of WSL programs and its FermaT specific AST syntax, Figure 7.1 represents the WSL program code of Listing 7.4.

```
1 IF x = 0 THEN PRINT("Goodby cruel world")
2 ELSIF FALSE THEN PRINT("Goodby cruel world")
3 ELSIF TRUE THEN PRINT("Hello World")
4 ELSE y := 2 FI
```

Listing 7.4: WSL Program: Hello World Example

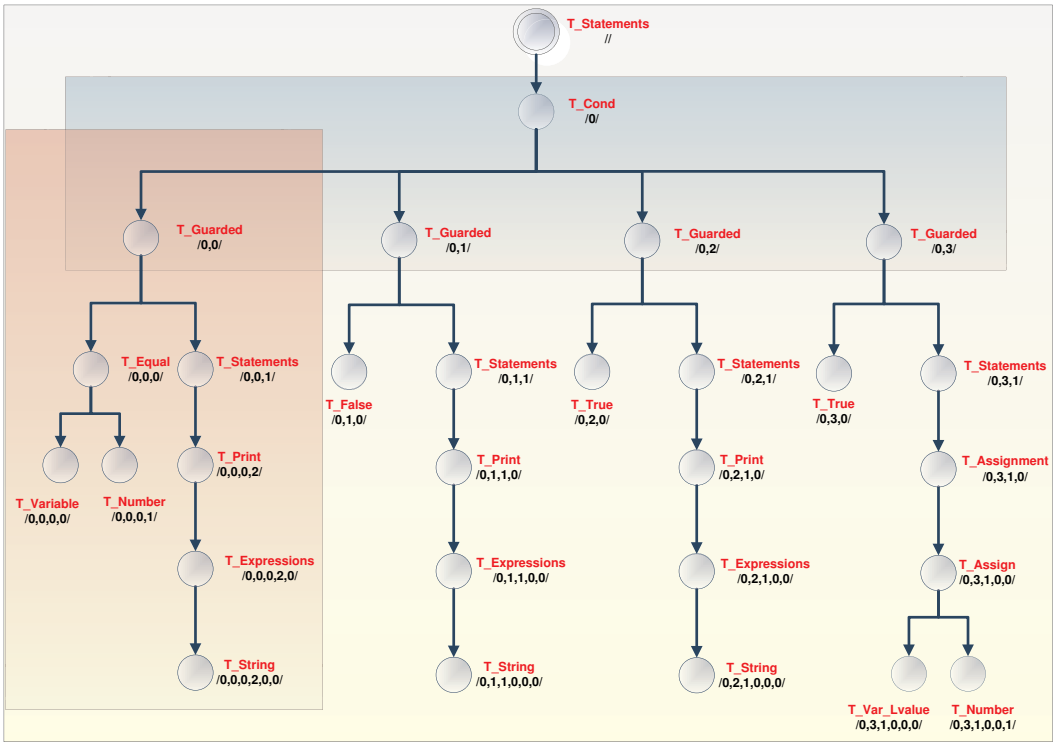


FIGURE 7.1: AST representing WSL Program: Hello World Example

7.2.2 FermaT Transformation Application

The following illustrates how FermaT transformations in combination with transformation scheme descriptions are applied. The transformation scheme description from Listing 7.2 serves as a demonstration.

Its definition is specified as follows: Transformation "*Simplify If*" with the Abstract AST path "*T_Cond*", (symbolised by the cited transformation) should be applied on the first WSL AST type "*T_Cond*" found within the program source. At first, this results in a WSL program search of Figure 7.1, and reveals that the AST node with the index "/0/" represents a WSL AST type "*T_Cond*". Applying the specified transformation results in the WSL program code presented in Listing 7.5. Its corresponding AST tree is illustrated in Figure 7.2.

```
1 IF x = 0
2   THEN PRINT("Goodby cruel world")
3   ELSE PRINT("Hello world") FI
```

LISTING 7.5: Hello World Example after the Transformation

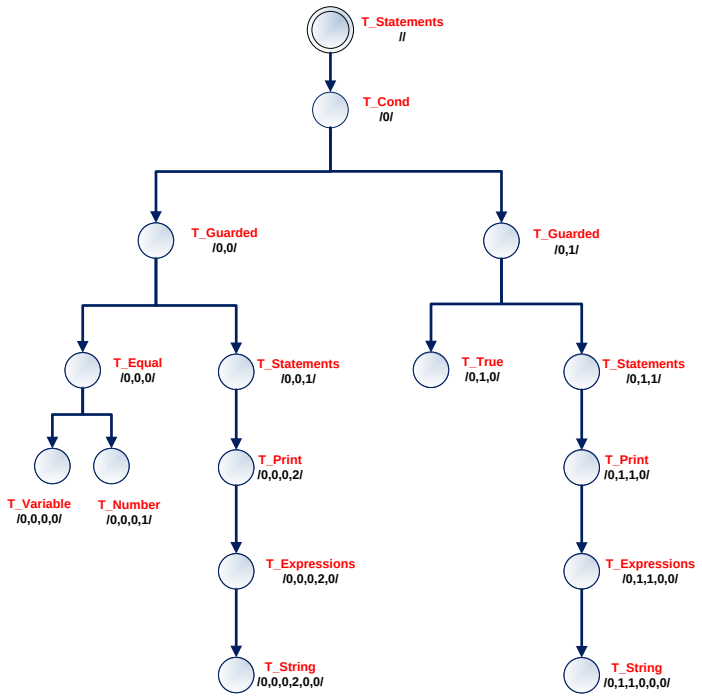


FIGURE 7.2: AST after Transformation within WSL Program: Hello World Example

The given example demonstrated the application of the FermaT transformation "*Simplify If*", which leads to the desired reengineering aim of "*simplifying the IF-Statement*" presented in Figure 7.2.

7.2.3 Transformation Sequence Application and Constraint Satisfaction

There are many ways within the application of WSL program transformations. The following example demonstrates how the application of a transformation sequence is satisfied. As the definition and application of transformation sequences is more complex compared to a single transformation, the maintainer's knowledge becomes a crucial aspect within their definition. If transformation scheme descriptions are not properly specified, the generation of a huge transformation sequence search space cannot be prevented. Processing this search-space would consume a lot of unnecessary processing time.

To present an illustration, the transformation scheme description in Listing 7.6 serves as a demonstration. The description specifies a sequence construct and consists of two transformations "*Simplify If*" and "*Simplify*" . The transformation "*Simplify If*" is further specified by a quantifier construct and states that the specified transformation

should be applied between "1" and "2" times on transformed WSL program source. This process leads to two different WSL program states which need to be analysed. The second transformation "*Simplify*" should only be applied on AST path "//" which is the AST root tree node.

```
1 (
2   < Simplify If @ T_Cond > [1 .. 2],
3   < Simplify @ // >
4 ) {C1}
```

Listing 7.6: Transformation Scheme Sequence Description

Transformations	Acronym
< Simplify If @ T_Cond >	T_0
< Simplify @ // >	T_1

Table 7.1: Substitution of the Transformations in Listing 7.6

With the transformation substitution in Table 7.1, the transformation scheme description can be specified as:

$(((T_0) \mid (T_0, T_0)), T_1)\{C_1\}$

Utilising the specified decomposition laws expressed and explained in Chapter 6, the following independent transformation sequences can be extracted.

sequence 1: $(T_0, T_1)\ \{C_1\}$
sequence 2: $(T_0, T_0, T_1)\ \{C_1\}$

Substituting the sequences with the original transformations results in:

sequence 1: (< Simplify If @ T_Cond >, < Simplify @ // >) $\{C_1\}$
sequence 2: (< Simplify If @ T_Cond >, < Simplify If @ T_Cond >,
< Simplify @ // >) $\{C_1\}$

After a brief analysis of the WSL program example in Listing 7.7, the following can be said about its WSL syntax structure: The WSL program source contains "2 IF-Statements" and therefore embeds "2 WSL *T_Cond* AST types". This reveals that the generated transformation sequences are applicable on "2 *T_Cond* FermaT AST" paths. This demonstrates that this process can be parallelised, submitting "sequence 1" to computing node "1" to compute the transformation sequence on AST path "1" and

the second submission, sending "sequence 2" to computing node "2" to compute the transformation sequence on AST path "2". At the end of this process, both results are evaluated according to the specified reengineering constraint "C_1".

```
1  k := 55;
2  j := k;
3  IF k < 50 THEN
4     j := j * 2;
5  ELSIF k < 25 THEN
6     j := j * 2;
7  FI;
8  IF k > 50 THEN
9     j := j * 1;
10 ELSIF k < 25 THEN
11    j := j * 2;
12 FI
```

Listing 7.7: WSL Program with 2x IF-Statements

In regard to this example, Figure 7.3 represents the internally created WSL transformation application tree. This tree is utilised to evaluate if a transformation sequence can be applied or not.

During each transformation process "T_i" the resulting WSL program "P_i" is tested if the reengineering constraint "C_1" has been fulfilled. If the transformation has been successfully applied but the constraint has not been fulfilled, the analysing process will continue with the next transformation. Otherwise the process of applying this transformation sequence stops and the next transformation sequence is chosen.

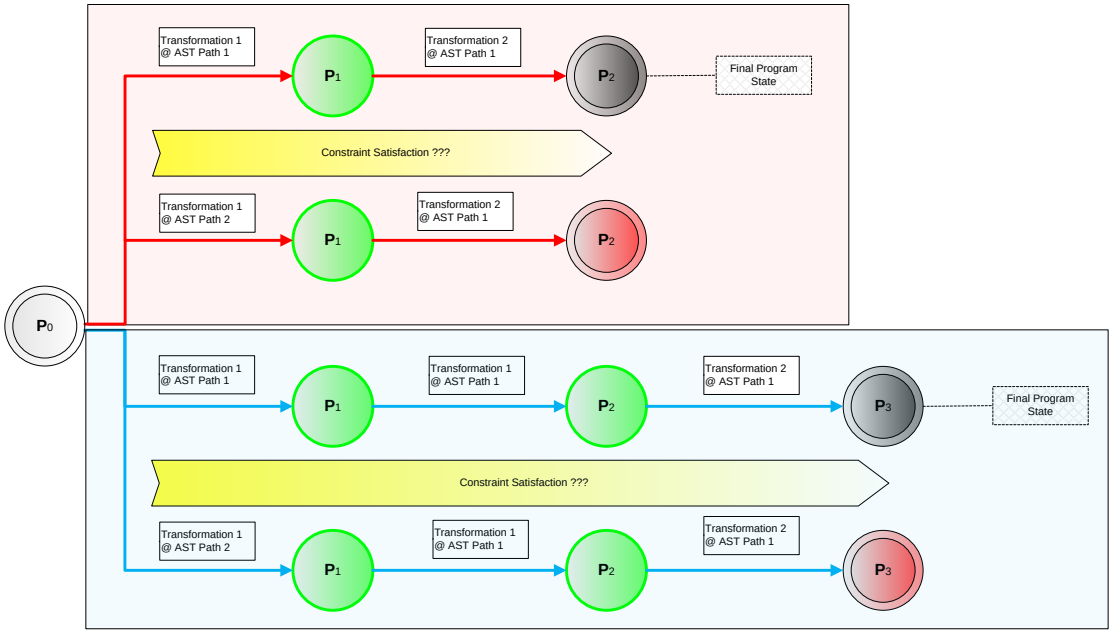


FIGURE 7.3: Analysis for the Application of the Transformation Scheme

7.2.4 Scheduling Parallel Transformation Tasks

The process of computing hundreds of transformation sequences can consume a lot of time. In order to speed-up parallel transformation tasks proper scheduling- and load-balancing-techniques need to be found, especially when the number of computing nodes within a parallel processing environment range from a single-digit number to hundreds. To exploit an efficient use of parallel environments proper resource management needs to be provided. The complexity can especially arise through the support of different physical architectures and communication infrastructures. Within this context, the proposed scheduling system combines communication and scheduling structures for parallel transformations processing. Special focus has been laid in the management of communication and scheduling at the same time. As scheduling is one of the concerns of finding an appropriate reengineering solution, heuristic solutions come into account. Figure 7.4 presents an overview of the developed scheduling system.

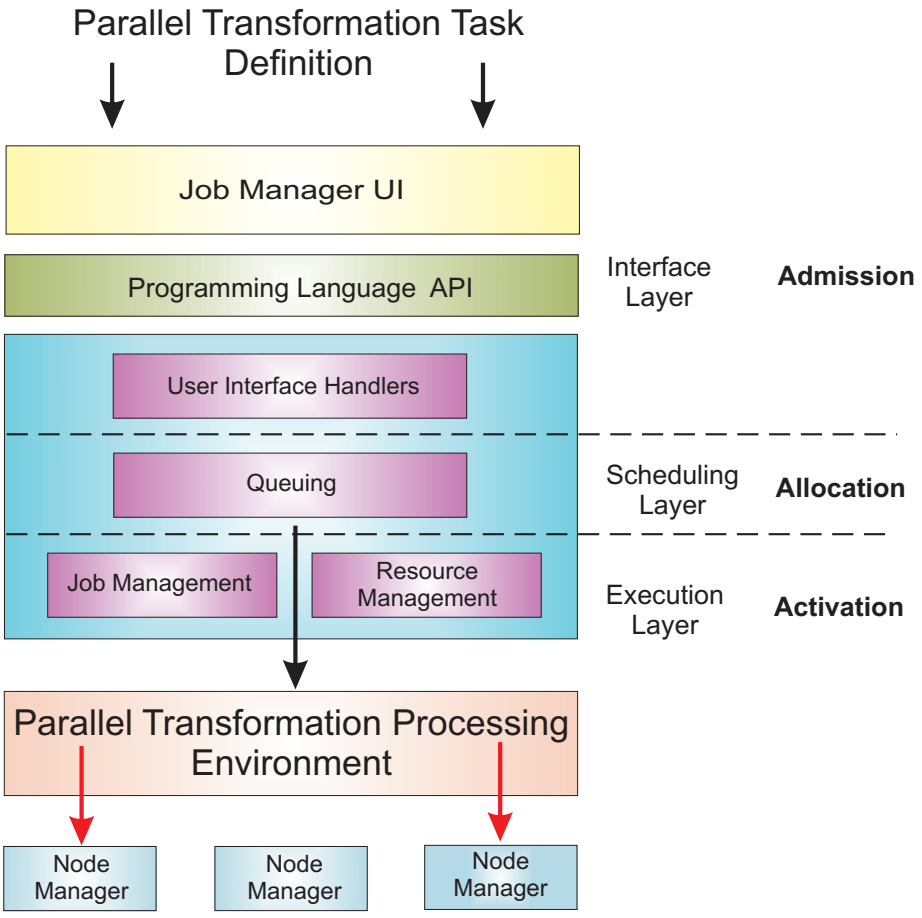


FIGURE 7.4: Parallel Transformations System Scheduling System Overview

As illustrated, the maintainer usually defines a parallel transformation task. Tasks are submitted through the headnode by the utilisation of the developed job-submission techniques described. Once a task has been assigned to the headnode, specified parallel transformations processing steps are evaluated. This process normally cannot be influenced by the maintainer unless the headnode service is going to be interrupted.

Within the next processing steps the headnode analyses the submitted parallel transformation task according to its specification. Based on this evaluation, the system starts queuing its parallel transformations processes. As commonly used, both static and dynamic task scheduling techniques are provided. Because the overall computing time is estimated by the headnode, static scheduling techniques are utilised. It can occur that a parallel transformations process takes longer than expected. For this reason the processing nodes are equipped with dynamic load balancing features. To fully present the objectives, the task scheduling process is divided into two categories: "**Static**" and "**Dynamic scheduling**".

- **Static scheduling** is the most common technique used within this approach. Within the static mode the headnode performs all calculations and task-submissions. This also includes the evaluation and utilisation of the presented decomposition laws to procedure transformation sub-scheme descriptions and transformation sequences. The generated sub-schemes are directly assigned to the compute nodes.

- **Dynamic scheduling** only occurs when transformations within a transformation scheme description are cited in the dynamic way. In this case the computing nodes utilise a technique to independently calculate how many transformations need to be computed and applied within the assigned WSL program. The headnode still filters inapplicable transformation sequences. Otherwise this would result in parallel transformations processing behaviour in which it could occur that transformation sequences are not applicable at all. This would slow down the system and waste parallel processing resources. To remedy this situation the scheduling-system always performs a first time analysis, before parallel transformation tasks are submitted to the environment. For a dynamic behaviour example, if a "*parallel constraint*" definition is not fulfilled within a given time frame, a computing node could send a signal indicating that it needs more computing time. Computing nodes can also acknowledge when they are finished with transforming.

Within this context the headnode always tries to estimate the overall computing time of each transformation task. The evaluation is based on the total number of applicable transformations within the transformation scheme description multiplied by the transformation processing times of each computing node. Once evaluated, the headnode calculates the workload balancing factor based on the number of evolved computing nodes and the possible parallelisation of the assigned parallel transformation task and transformation scheme descriptions. As this process offers many variations of parallelisation, the headnode first searches for alternative-constructs within the transformation scheme description. Additional parallelisation can be evaluated within the AST path specification of transformations within a transformation scheme description.

The headnode simply parallelises transformation tasks by the utilisation of the "*divide-and conquer*" algorithm. Because it could occur that the assigned task takes longer to be computed, due to the fact that transformations inside the transformation scheme description are expressed in a dynamic manner, the computing nodes can utilise the heartbeat-communication system to indicate how far they are within the assigned task.

7.3 Parallel Transformations Processing

The following sections outline how parallel transformations tasks can be parallelised. Previously gained knowledge about successful program transformation application within the FermaT transformation system is now transferred to the parallel domain. The following parallel transformations processing techniques are utilised and introduced within this thesis, "*parallel-*" and "*linear- line*" transformations processing:

- **Parallel Transformation Processing:**

  The possibility of assigning a parallel transformations task to a parallel architecture has been discussed within an earlier state. To guide a parallel process, parallel constraints are introduced. On the basis of the complexity of assigning transformation tasks, transformation scheme descriptions and parallel constraints to the headnode, the system automatically evaluates a suitable parallel processing technique. Hence, this has to be distinguished between the two processing modes, automatic and manual:

  - Within the automatic mode, parallel transformation task are processed without any human interaction. With the guidance of parallel constraints, this process either succeeds or fails.
  - Within the manual mode, the maintainer manually designs and assigns a parallel transformation process to the system, by specifying WSL program files, transformations or transformation scheme descriptions manually.

- **Linear Array Processing** Within this mode, the specified parallel architecture functions as an architecture of multiple parallel linear-lines, processing tasks similar to an array. Computing nodes are linked to a linear-line where each line processes transformation tasks independently. This technique opens the opportunity to process more than one WSL program at a time. Another advantage of this technique is that different WSL programs can be assigned to a specific transformation sub-scheme description. A sub-scheme in this case could specify a specific reengineering goal.

7.4 Parallel Transformations Processing Design

As highlighted in Chapter 4, different parallel processing architectural designs are utilised within this parallel transformations processing context. Furthermore as illustrated in

Chapter 6, transformation scheme description decomposition laws are introduced to capture parallelism. Transformation scheme descriptions in this case are considered as the processing roadmap of a transformation task, leading to a specified maintenance goal. The extracted transformation sequences should lead to the reengineering aim. This been explained in Section 7.2. Transformation scheme description decomposition laws produce sequences of transformations. Since each transformation sequence is unique in its specification, they can be considered as independent and therefore can be processed in parallel. To compute hundreds of transformation sequences in a dynamic parallel way, different architectural constellations need to be evaluated and utilised.

For example, the opportunity to choose between different parallel techniques opens the possibility to add computing nodes dynamically during runtime to the system. This technique allows for an ad-hoc speed-up of the transformation processes. The following outlines how transformation scheme descriptions are mapped to a parallel transformations processing environment, by the provision of the above described techniques.

7.4.1 Parallel Transformations Processing

Before transformations, transformation scheme descriptions or transformation sub-scheme descriptions can be assigned to computing nodes they need to be analysed. This processing step is performed by the analysing system. Listing 7.8 represents a sample transformation scheme description. Analysing and decomposing this schema according to the defined laws, reveals that the presented description inherits "*four transformation sequences*".

```
1 (
2   < Simplify If @ T_Cond > [1 .. 4],
3   < Simplify @ // >
4 ) {C1}
```

Listing 7.8: Simple Transformation Scheme Sequence Description

Transformations	Acronym
< Simplify If @ T_Cond >	T_0
< Simplify @ // >	T_1

Table 7.2: Substitution Transformations

$(((T_0) \mid (T_0, T_0) \mid (T_0, T_0, T_0) \mid (T_0, T_0, T_0, T_0)), T_1)\{C_1\}$

By the utilisation of the specified laws expressed in Chapter 6, it reveals that four transformation sequences are embedded within the quantified construct. For simplicity, the transformations are substituted by the above stated Table 7.2.

sequence 1: $(T_0, T_1)\ \{C_1\}$
sequence 2: $(T_0, T_0, T_1)\ \{C_1\}$
sequence 3: $(T_0, T_0, T_0, T_1)\ \{C_1\}$
sequence 4: $(T_0, T_0, T_0, T_0, T_1)\ \{C_1\}$

Before the transformation sequences can be mapped, an analysis of the parallel transformations processing environment needs to be performed. Figure 7.5 presents a parallel environment sample.

This architectural concept contains the following components: a headnode with file and database services, a terminal to enter and submit parallel transformation tasks and four computing nodes. Four computing nodes are chosen to equally match the number of generated transformation sequences.

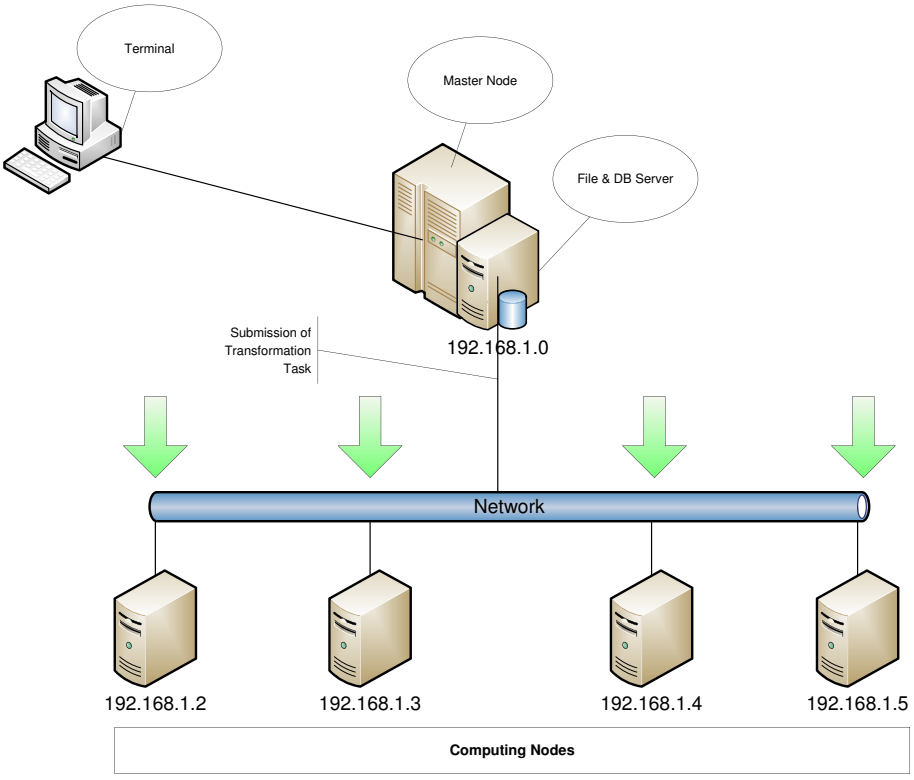


FIGURE 7.5: Parallel Transformation Processing Design

On the basis of the specified parallel transformations processing and scheduling algorithms, the headnode analysing system automatically assigns each computing node one transformation sequence. This mapping is illustrated in Table 7.3.

IP Address	Transformation Sequence
192.168.1.0	(Headnode)
192.168.1.2	Sequence 1
192.168.1.3	Sequence 2
192.168.1.4	Sequence 3
192.168.1.5	Sequence 4

TABLE 7.3: Computing Node Transformation Sequence Assignment

Once evaluated and visually mapped to the compute nodes, each transformation within the generated transformation sequences is substituted by its FermaT transformation name. The following lines illustrate this procedure.

Sequence 1: (< Simplify If @ T_Cond >, < Simplify @ // >) $\{C_1\}$
Sequence 2: (< Simplify If @ T_Cond >, < Simplify If @ T_Cond >,
< Simplify @ // >) $\{C_1\}$
Sequence 3: (< Simplify If @ T_Cond >, < Simplify If @ T_Cond >,
< Simplify If @ T_Cond >, < Simplify @ // >) $\{C_1\}$
Sequence 4: (< Simplify If @ T_Cond >, < Simplify If @ T_Cond >,
< Simplify If @ T_Cond >, < Simplify If @ T_Cond >,
< Simplify @ // >) $\{C_1\}$

Further processing includes the encapsulation of the above stated transformation sequences into a "*PLACED PAR*" construct. This construct is needed for the computing node assignment.

PLACED_PAR
192.168.1.2 TCP (1, WSL FileName, **Sequence 1**)
192.168.1.3 TCP (2, WSL FileName, **Sequence 2**)
192.168.1.4 TCP (3, WSL FileName, **Sequence 3**)
192.168.1.5 TCP (4, WSL FileName, **Sequence 4**)

As a demonstration for this example, the TCP/IP communication layer and some random values were chosen. The WSL filenames represent the starting WSL program source "P_0".

7.4.2 Linear Array Parallel Transformation Processing

The linear-line transformations processing differs from parallel transformations processing in the way that transformations, transformation schemes or transformation subschemes are assigned to individual computing nodes. Figure 7.6 gives an illustration

on how a parallel linear-line array architecture could be designed based on the transformation scheme description outlined in Section 7.4.2. Depending on the number of computing nodes involved, nodes are aligned in parallel. This procedure is performed by utilising the developed parallel communication constructs. By this means computing nodes are instructed which neighbours they need to communicate with. Within the given example, the first computing node line of the "*linear-line array*" would be connected to the following:

Linear Array Line 1
Computing Node 1 (192.168.1.2)= Neighbor A: 192.168.1.0 & Neighbor B: 192.168.1.3
Computing Node 2 (192.168.1.3)= Neighbor A: 192.168.1.2 & Neighbor B: 192.168.1.0

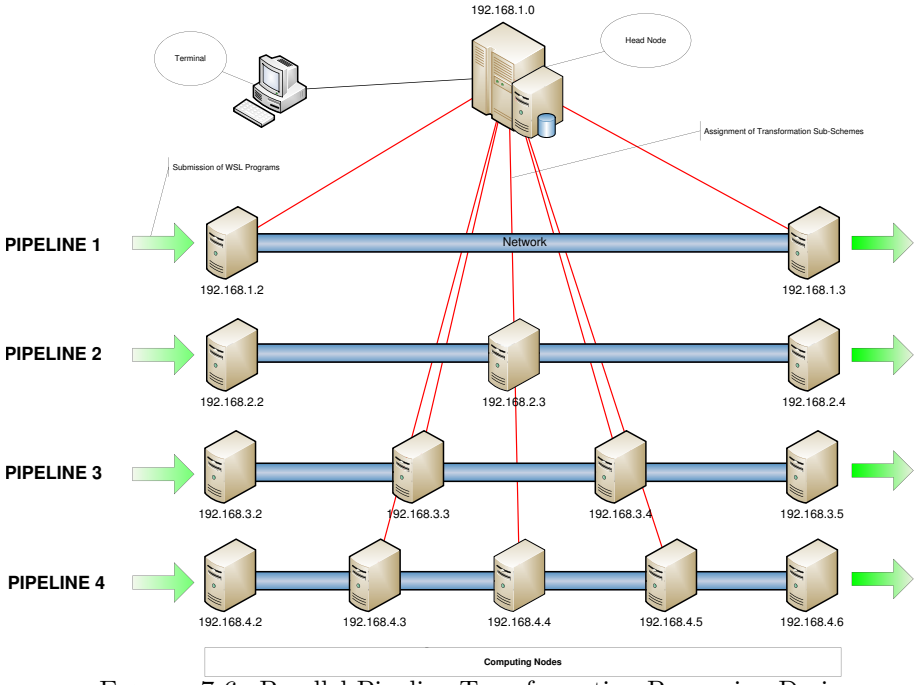


FIGURE 7.6: Parallel Pipeline Transformation Processing Design

Once the parallel architecture has been laid out, the extracted transformation sequences are assigned to each linear line demonstrated in Table 7.4.

Pipeline	**Transformation Sequence**
Linear Array Line 1	Sequence 1
Linear Array Line 2	Sequence 2
Linear Array Line 3	Sequence 3
Linear Array Line 4	Sequence 4

TABLE 7.4: Transformation Sequence Pipeline Assignment

Within the next processing step, each transformation sequence will be further separated by its FermaT transformations. Each will be assigned to one computing node, listed in Table 7.5. As a FermaT transformation is an individual process which produces a new WSL program state, it can be considered as an independent step and therefore mapped to the developed parallel processing environment.

Pipeline	**Node 1**	**Node 2**	**Node 3**	**Node 4**	**Node 5**
Linear Array Line 1	T_0	T_1 $\{C_1\}$			
Linear Array Line 2	T_0	T_0	T_1 $\{C_1\}$		
Linear Array Line 3	T_0	T_0	T_0	T_1 $\{C_1\}$	
Linear Array Line 4	T_0	T_0	T_0	T_0	T_1 $\{C_1\}$

TABLE 7.5: Transformation Pipeline Assignment

As a result, line one of the "*linear-line array*" could be described by the "*PLACED PAR*" construct as:

PLACED_PAR

Node 1 TCP (1, WSL FileName, T_0)
Node 2 TCP (4, WSL FileName, $T_1\{C_1\}$)

As another example, line three of the "*linear-line array*" could be outlined with a "*PLACED PAR*" construct as:

PLACED_PAR

Node 1 TCP (1, WSL FileName, T_0)
Node 2 TCP (2, WSL FileName, T_0)
Node 3 TCP (3, WSL FileName, T_0)
Node 4 TCP (4, WSL FileName, $T_1\{C_1\}$)

Once all transformations have been assigned and confirmed by the computing nodes, the specified WSL program source "P_0" is passed to and computed individually by each linear line.

This process usually starts from the left and is illustrated in Figure 7.6. To verify the reengineering constraint ("C_1") the last computing nodes of each processing line evaluate and pass the result to the headnode.

By the identification of which reengineering direction a specified transformation scheme description leads, resulting sub-scheme descriptions can be reused to process more than one WSL program at a time.

7.5 Evaluation of the Computation Time

One of the key aspects of this approach is the analysis of the overall processing time. Its evaluation is crucial as the maintainer needs to be notified, if a parallel transformation task can be satisfied according to its definition or not. Its calculation is usually an evaluation of the combination of transformation scheme analysis and computing power of the parallel environment. Task embedded parallel processing constraints specifications which outline parallel transformation process and are relevant for this calculation process, do not have any effect on the transformation scheme embedded reengineering aims or vice versa.

On the basis of each transformation process specification, the maintainer can be notified if a task can be fulfilled or not. However each task satisfaction also has to do with its specification and the utilisation of parallel processing constraints. As an example, if the maintainer likes to process a transformation task with 8 computing nodes and 6 computing nodes are currently only available within the system, he will be notified that the specified task can be only computed with 6 work nodes. To give a general overview of this computing time algorithm, the following can be said:

- Each transformation scheme description or transformation sequences are decomposed within the headnode to evaluate possible parallelisation.
- After the elimination of redundant and unsatisfactory transformation sequences, the overall computing time can be estimated.
- Each transformation within a transformation scheme or transformation sequence is measured with a computing time of one second.
- Overall computing time estimate is presented to the maintainer before a transformation task is computed.

To give an example, the following can be said about the specified alternative-construct cited below. It is further assumed that all transformations within this expression are applicable.

$((T_0 \mid T_1 \mid T_2 \mid T_3)[1 \,..\, 3]\,)\{C_1, C_2\}$

During the first unrolling and transforming process there exist at least four different WSL program states "P_{1-4}". The next processing step results in a combination of "4 x 4 transformations" resulting in 16 transformations. Another decomposition step further, "16 x 4 transformations" generate 64 different WSL program states. In this case the time-algorithm assumes that there are at least 64 different program states "P_{1-64}".

However it has to be kept in mind that this number can vary based on the cited AST path or on the application of each FermaT transformation on more than one AST tree path within the WSL program source. Due to the NP-problem of the generated search-space and the applicability of the transformations, these values only serve for estimation purposes to evaluate the overall processing time. The case studies in Chapter 9 reveal that these numbers can vary quite considerably.

7.6 Transformation Task Evaluation

As previously outlined, the successful computation of a parallel transformation task is measured by the satisfaction or none satisfaction of the overall reengineering constraints embedded within a transformation scheme description. To assist each parallel transformation process, the developed parallel transformation processing language can be utilised to guide and direct individual processes. However, the definition and the embedding of parallel processing constraints generally used to accelerate the overall transformation process do not limit or restrain the overall satisfaction of the reengineering process. This is due to the fact that the task specific parallel computation refinements do not have any influence on the specified reengineering constraints or vice versa. This is indispensable because both constraints categorisation need to be independent to be able to achieve the parallelisation process. Thus to ensure that every parallel transformation task is computed by the parallel processing architecture, it has to have no restrictions. Nonetheless the maintainer is always notified beforehand if a specified parallel transformation task can be satisfied according to its refinements of not. However the case studies demonstrate how crucial it is to properly define a transformation task and its reengineering constraints. Generally speaking, the more a transformation task

is restricted by its transformations and constraints, the more the reengineering success can be guaranteed.

7.7 Summary

This chapter presented how FermaT program transformations are applied. It demonstrated how transformation sequences can be computed. It also showed how parallel transformation tasks are mapped to a parallel processing environment and how its transformation sub-schemes and transformation sequences are treated. At the end, examples present how transformation processes can be estimated and evaluated within the demonstrated parallel transformations environment.

Chapter 8

Prototype Tool Support

Objectives

- To provide an overview and description of the FermaT Cluster Environment (FCE).
- To outline the FCE's architectural components.
- To present the implementation of the parallel transformations framework.
- To illustrate the FCE's integration within the FermaT transformation system.

8.1 Introduction

This chapter reviews and describes the FermaT Cluster Environment (FCE), a prototype tool, developed to support the parallelisation of transformations within FermaT transformation system. The tool verifies the proposed thesis, wherein its description and implementation is presented. At the end, the FCE's Graphical User Interface (GUI) demonstrates the features and capabilities of the outlined parallel transformations framework.

8.2 The FermaT Cluster Environment (FCE)

At the beginning of the FCE's development process, the FermaT Maintenance Environment (FME) [64] served as a guideline. Both tools need to address the application of program

transformation within the FermaT transformation system. However, most of its FermaT addressing components had to be lightened and enhanced to address parallelisation. On the other hand, some completely new features had to be designed and implemented. To support parallel transformations processing within FermaT, the following use cases are identified:

- Establishment of an automated parallel transformations processing infrastructure.
- Possibility to manually define parallel transformation tasks, followed by manual assignment to compute nodes.
- Development of a scheduling and parallel processing system, controlled via constraints and user interaction.
- Implementation of a Graphical User Interface (GUI) to control illustrated features with additional support of:
  - Navigation through WSL program source.
  - Application of program transformations.
  - Command line FermaT engine navigation.
  - Extraction of program graphs.
  - Parallel transformation task definition, submission and evaluation.
  - Parallel transformations system control features.

The approach of using transformation sequences in combination with constraints to guide and fulfil a maintenance goal has demonstrated [6], that parallel computing power is strongly needed to compute transformation tasks in reasonable time. This supported the analysis of the existing FermaT transformation system to detect parallelisation possibilities. After its evaluation, the following solution seemed to be the most appropriate, to fulfil the need for an automated parallel transformations processing system:

- The use of a Beowulf style computer cluster by utilising standard Commercial Off-The-Shelf (COTS) PC components for parallel transformations processing.
- Utilisation of open-source software systems such as Linux, Message-Passing-Interface (MPI) libraries and Secure Shell (SSH) access tools to develop and implement a parallel processing infrastructure.
- Employment of the Ethernet standard as a communication base, plus the combination of the TCP/IP and SSH protocol stack, to address computing nodes in a directly and key-less manner.

- Development of a GUI to control FCE's parallel features as:
  - Parallel environment configuration possibilities.
  - Controlling and monitoring of parallel processing steps and computing node behaviour.
  - Assignment of parallel transformation tasks directly to computing nodes.
  - Computing node failure and recovery management through integrated communication implementations, followed by remote access and starting features.

The classical Beowulf cluster style architecture, consisting of independent homogeneous computing nodes, seemed to be the most appropriate solution in the matter of parallel transformations processing. It turned out to be the cheapest way of establishing a parallel processing environment, whereas SMP systems are far more expensive [61]. In regard to the programming language, Java has been chosen to be the most appropriate choice, due to its platform independence. The possibility to integrate the parallel environment into the FermaT Maintenance Environment (FME) also contributed to that choice, whereas the connection to other programming languages such as C could be facilitated through the Java Native Interface (JNI). FCE's features, both for local- and parallel- transformations processing can be used either on MS Windows or Linux based platforms.

How the parallel transformations processing environment can be set-up to successfully compute transformation tasks in parallel, is illustrated within a separate available FCE tutorial. The following sub-sections outline in more detail how this environment functions and which techniques are utilised to realise and satisfy the proposed functions.

8.3 A Beowulf Architecture for Parallel Transformation Processing

In the matter of parallel processing and in terms of cost and efficiency the Beowulf style cluster architecture seemed to be an appropriate solution. This decision was eased through the Personal Computer (PC) related mass market of commodity computing systems and the availability of open source software components. Rather than a fixed single system of parallel computing devices such as SMP systems, this parallel architecture presents a system that demonstrates the evolution of commodity hardware in combination with today's open source software components [4].

The presented parallel transformations processing Beowulf style cluster can be accessed via any standard PC or laptop. To be able to directly submit parallel transformation

tasks to the environment, the user needs to have access to a specified NFS directory. This feature is mandatory because parallel transformation task underpinning files need to be accessible by all computing components. The established prototype parallel transformations processing environment consists of the following hardware components:

- 1 Intel Pentium IV 2,5 GHz PC, functions as the headnode, NFS- and database-server.
- 6 Intel Pentium III 933 MHz PCs, with 264 MB of Random Access Memory (RAM) and equipped with a 20GB Hard Disk Drive (HDD), function as computing nodes.
- Ethernet equipment such as a Wireless Local Area Network (WLAN) Router, LAN s switches and network cables connect the hardware to a parallel processing environment.

Figure 8.1 illustrates the presented parallel hardware architecture.

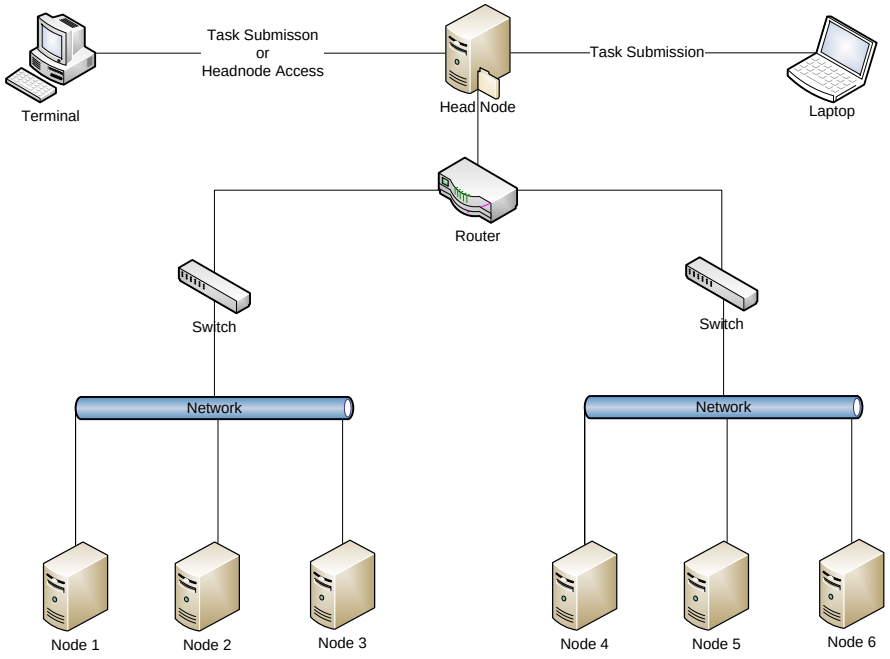


FIGURE 8.1: FermaT Cluster Environment (FCE) Network Architecture

Computing nodes store their local processing data on their HDD. Each computing node is attached to the switch. A switch was chosen to reduce network traffic. The exchange of processing results is realised through the use of the messaging system outlined. Transformation tasks can be assigned to the environment by a terminal or laptop.

8.4 The Network File and Communication System

To successfully compute transformation tasks, computing nodes need to share and exchange processing data. How they exchange the data is a definition of the specified transformation task, utilising the formal language explained in Chapter 4.6. Commonly used data needs to be in an accessible directory within the environment. This is realised by specifying an overall NFS directory located within the system's file server. In the default configuration, this folder is located within the headnode. How these folder structures are set-up is outlined within the separate available FCE tutorial.

To facilitate the recognition of computing nodes, a Dynamic Domain Name Service (DynDNS) is utilised. This service opens the possibility to access work nodes via a unique identifier name. For example, the headnode of the environment can always be accessed via the name "*fc-head.homeip.net*". This feature guarantees to always have access to the headnode and their computing nodes, also when their IP address changes due to a new Dynamic Host Configuration Protocol (DHCP) server configuration. In order to connect and access computing nodes from the headnode without interfering with any other protocol restrictions, the Secure Shell (SSH) protocol is utilised. This opens the possibility to access computing nodes in a secure and direct manner. A particular activity is conducted to avoid any password exchange every time the headnode or computing node exchange processing data. For parallel processing communication facilities, TCP/IP sockets and Java's RPC functions are utilised.

In order to start a computing node remotely special scripts are needed. These scripts are executed during the environment start-up process. A script executed within a computing node is listed below, specifying the following:

```
java -classpath ./lib/fce-node.jar:./lib/xercesImpl.jar:
./lib/xmlParserAPIs.jar
-Djava.rmi.server.hostname=fc-node2.homeip.net
fme.components.highperformancecomputing.Client
fc-head-xps420 fc-node2
```

With the following refinement:

- **java -classpath:** Specifies the directory of the runtime libraries.
- **-Djava.rmi.server:** Specifies the DNS name or IP address of the RMI server.
- **fce.components.computeNode fc-head-xps420 fc-node2:** States the classpath for the computing node, followed by the DNS name or IP address of the headnode and the computing node identification name.

These scripts are remotely called by the headnode; subsequently each computing node automatically starts an instance of the FermaT transformation engine. This procedure usually follows an analysis of the computing node's transformation processing speed, by executing special developed scripts.

8.5 The Computing Node Analysis

Through the development of transformation processing analysis scripts, the headnode is able to judge how fast computing nodes can manage their processing load. To evaluate their transformation processing time, each FermaT transformation is executed and their application time is recorded. These scripts assist the calculation process of finding an appropriate computing task load-balance. They are usually executed once, during the computing nodes registration process, unless the computing node hardware configuration changes. It only takes a couple of seconds to evaluate the processing speeds and address them to the headnode. Their outcome is specified within an Extensible Markup Language (XML) based file and Listing F.1 presents an example. Listing 8.1 provides the specification for the FermaT transformations performance test.

```
<FermaT_Transformations>
  <Group>
     <Name>Group Delete</Name>
     <Transformation>
          <Name>Delete All Redundant</Name>
          <FermaT_Engine_Name>//T/R_/Delete_/All_/Redundant
          </FermaT_Engine_Name>
          <WSL_Test_File>delete_all_redundant_example_1.wsl
          </WSL_Test_File>
          <AST_Path>null<
          /AST_Path>
     </Transformation>
  </Group>
</FermaT_Transformations>
```

Listing 8.1: FermaT Transformation Performance Test

With the following attributes:

- **FermaT_Transformations:** Introduces the FermaT Transformation Performance Test.
- **Group:** Each FermaT transformation belongs to a specific group, this type introduces each transformation group.
  - **Name:** Addresses the name of the group.
  - **Transformation:** Introduces a specified FermaT transformation, followed by additional attributes:
    * **Name:** General name used to identify the transformation.
    * **FermaT_Engine_Name:** The FermaT transformation engine name, internally used to call and address the transformation.
    * **WSL_Test_File:** The WSL test file name which should be used to evaluate each transformation processing speed. This file is located within a specified NFS folder.
    * **AST_Path:** Specifies the AST path on with the transformation should be applied.

The test results in a XML based transformation processing speed evaluation file. An example of Computing Node "*192.168.1.72*" and its hardware configuration is presented on the next page.

```
1  <FermaT_Trans_Performance_Test>
2       <Node_IP>192.168.1.72</Node_IP>
3       <Date_Stamp>2010_02_08</Date_Stamp>
4       <Time_Stamp>14:57:03</Time_Stamp>
5       <CPU>PIII_933</CPU>
6       <RAM>264_MB</RAM>
7       <HDD>20_GB</HDD>
8       <Transformation>
9            <FermaT_Engine_Name>//T/R_/Delete_/All_/Redundant</FermaT_Engine_Name>
10           <Processing_Time>5</Processing_Time>
11      </Transformation>
12      <Transformation>
13           <FermaT_Engine_Name>//T/R_/Remove_/All_/Redundant_/Vars</
     FermaT_Engine_Name>
14           <Processing_Time>5</Processing_Time>
15      </Transformation>
16      <Overall_Time>
17           <Processing_Time>138</Processing_Time>
18      </Overall_Time>
19      <Transformation>
```

```
20            <FermaT_Engine_Name>//T/R_/Merge_/Right</FermaT_Engine_Name>
21            <Processing_Time>5</Processing_Time>
22       </Transformation>
23  ....
24       <Overall_Time>
25            <Processing_Time>286</Processing_Time>
26       </Overall_Time>
27  </FermaT_Trans_Performance_Test>
```

LISTING 8.2: FermaT Transformation Performance Test Results

- **FermaT_Trans_Performace_Test:** Introduces the FermaT Transformation Performance Test.
- **Node_IP:** Specifies which computing node was tested.
- **Date_Stamp:** Tags the performance test with a date. This information is used to reschedule tests on a frequently basis.
- **Time_Stamp:** Tags the test with a time. The headnode uses this information to record the communication time delay between sending and receiving this message from the computing node.
- **CPU speed:** Expresses the CPU speed of the computing node.
- **RAM:** Specifies the RAM size of the node.
- **HDD:** Addresses the HDD space of the computing node.
- **Transformation:** Introduces the FermaT transformations used.
  - **FermaT_Engine_Name:** States the FermaT transformation engine name.
  - **Processing_Time:** Cites the computation time of the specified transformation in milliseconds.
- **Overall_Time:** Expresses the overall computing time of all specified transformations.
  - **Processing_Time:** States the Overall transformation processing time in milliseconds.

NOT ALL FERMAT TRANSFORMATIONS ARE LISTED WITHIN THIS GIVEN EXAMPLE, IT JUST SERVES AS A DEMONSTRATION SAMPLE!

8.6 FermaT Transformation Engine (FTE) Integration

The FermaT transformation engine is a command line based tool. The connection between the transformation engine and the FCE is realised through software pipes. As Java supports a possibility to start OS processes pipes can be utilised. To run and command the transformation engine, basic instructions are combined within these scripts and run under Linux and MS Windows OS systems. In order to cover and recognise FermaT transformation engine operating commands: read-, write- and error streams are designed. Their design is presented in Figure 8.2.

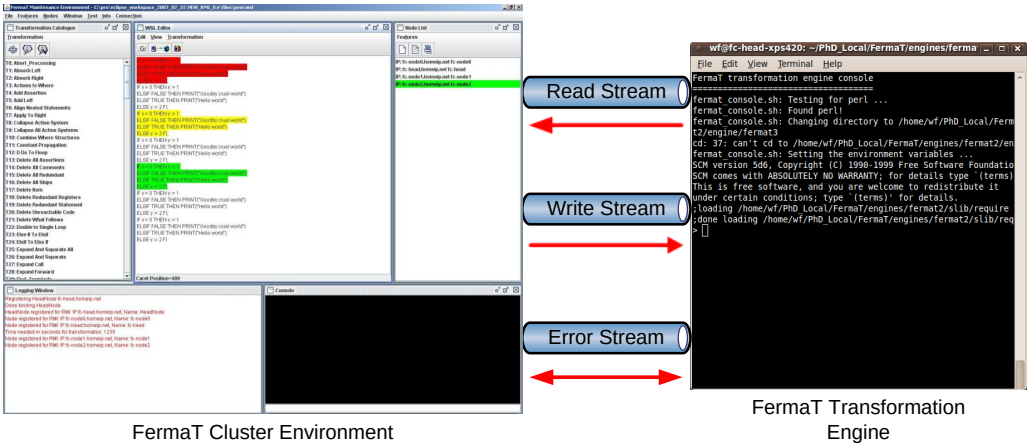


FIGURE 8.2: Pipe Connection between FermaT and the FCE

The limitation of processing speed was solved by a careful stream design. This has been solved by submitting sequences of transformations at once in one direction, before operating with a stream of the opposite direction. Therefore the processing delay could be minimised. This was proven to be true during case studies analysis, and demonstrated that a transformation process takes 0.2 second to be processed by computing more than 38.000 transformations in less than 2 hours.

8.7 The FermaT's Basic Control Commands

The free available FermaT transformation engine, "*fermat3*", is published under GPL and can be obtained from the internet. Once extracted from the web the system can be immediately executed, as the system includes all necessary binaries to run under Windows or UNIX/Linux OS. The FermaT transformation engine can be simply started

by running the "*fermat console.sh*" command under Linux or by executing "*MinGW-scmfmt.exe*" under Windows. This launches the Scheme interpreter and opens for the user to enter commands right at the command line. Commands are immediately translated and applied to the FermaT transformation system or on its loaded WSL programs. FermaT's most important processing commands are listed below.

(@New_Program (@Parse_File "test.wsl" //T_/Statements))

Before transformations can be applied, a WSL program has to be loaded into the transformation engine. This can be performed by the command above. Scheme is a dialect of LISP and it is case sensitive. All capital letters within a command need to be lead by a "/". All WSL symbols need to be preceded by "/", to avoid clashes with Scheme symbols.

Once the command has been passed to the FermaT engine, the quoted WSL file is parsed by its engine and internally represented as an Abstract Syntax Tree (AST). Each AST node within a WSL program tree records a specific type of node. This can be either the value stored (null or empty value) or the sequence of components of the node. For example, an "*IF-Statement*" as specified by the WSL AST type node "*T_Cond*" can have any positive number of components of the type "*T_Guarded*". Based on the WSL syntax specified in Appendix B.7, this must include two components: a condition and a statement sequence. The statement sequence can have any positive number of components of the generic type "*T_Statement*".

A WSL program tree can be extracted from the engine via the following command. This simply prints the AST in a top-down manner.

(@Print_WSL (@Program) " ")

In order to apply a transformation on a specific AST node, the engine has to know on which node the transformation should be applied. This position can be reached by traversing through the tree while using the commands below. The "*@GOTO*" command can be utilised to directly go to a position. These FermaT specific commands are reused by transformations to check if the transformed WSL program constructs correspond to their specification.

- @Left: Move to the previous component at the current level.
- @Right: Move to the next component at the current level.
- @Up: Move up the tree to the parent item.
- @Down: Move down to the first component of the current item.

- @To(n): Move to the nth component at the current level.
- @Down_To(n): Same as: @Down; @To(n).
- @To_Last: Move to the last component at the current level.
- @Down_Last: Same as: @Down; @To_Last.
- @GOTO: Jumps to a specific AST node position.
- @POSN: Returns the current position within the AST, as a sequence of integers (indices), starting at the root and working down the tree.

With these commands, the FermaT parse tree can be totally traversed. The function "*@Trans*" is used to apply a program transformation on a AST node. Each FermaT transformation has to fulfil a applicability condition. For example, the transformation "*Simplify If*" would be suitable on the specific AST type "*T_Cond*":

(@Trans //T/R_/Simplify_/If)

The transformation "*Simplify If*" is only applicable on this specific AST node "*T_Cond*" and simplifies an "*IF-Statement*". The "*@Trans*" function can be also used with an optional string parameter, which can contain additional processing data. If a transformation has been successfully applied, the FermaT engine acknowledges this by returning a charter "#t". sequence. In case of failure the engine would print an error message and return "#f". In order to avoid failures during a transformation process resulting in unpredictable results, it is advisable to test the applicability of each transformation. This can be tested with the "*@Trans?*" function. After a transformation has been successfully applied, the modified program can be written to a file by utilising FermaT's WSL Pretty Printer. The following command performs this step:

(@PP_Item (@Program) 80 "example_result.wsl")

FermaT would parse its engine AST, pretty prints it into readable WSL source code and writes it to the file "*result.wsl*".

In order to evaluate different metrics defined within a constraint to satisfy a maintenance goal, the following FermaT commands can be utilised:

(@Posn): Returns the current position within the AST.

(@Total_Size(@I)): Returns the total number of AST nodes (item) at "@Posn I".

(@McCabe_FCE (@I)): Expresses the McCabe Cyclomatic Complexity (CC) measure for "@Posn I".

(@CFDF_Metric_FCE (@I)): Returns the Control-flow / data-flow metric on "@Posn I".

(@What_Trans_FCE (@I)): Returns the total number of applicable transformations at "@Posn I".

8.8 FermaT Transformations and their Capabilities

Before the FCE can be utilised to start parallel transformations processing on the basis of the FermaT transformation engine, valuable information about each transformation needs to be extracted. All FermaT transformations are located in FermaT folder "*/src/-trans*". Each transformation is specified via a description pinned "**d.wsl*" and a source "**.wsl*" file. The description file includes information about each transformation such as its name, keywords and description of what it does. Each source file includes the source of its transformation and internal procedures to test and apply it.

The example below illustrates the source file for the "*While to Floop*" transformation. Analysing the specified $\mathcal{META}$WSL "MW PROC @While To Floop Test()" function reveals that the transformation is only applicable on the AST type "*T_While*". Otherwise the FermaT engine would acknowledge the applicability request with a failure: "*@Fail*". The second $\mathcal{META}$WSL procedure specifies and transforms the WSL "*While*" structure into a "*DO-Loop*" based on its specification.

```
MW_PROC @While_To_Floop_Test() ==
  IF @ST(@I) <> T_While
    THEN @Fail("Selected item is not a WHILE loop.")
    ELSE @Pass FI;


MW_PROC @While_To_Floop_Code(Data) ==
  VAR < B := < >, S := < > >:
  IFMATCH Statement WHILE ~?B DO ~*S OD
  THEN B := @Not(B);
       @Paste_Over(FILL Statement DO IF ~?B THEN EXIT(1) FI; ~*S OD ENDFILL)
  ELSE ERROR("Not a WHILE loop!") ENDMATCH ENDVAR;
```

Information about the application of each transformation is important for the FCE, due to the fact that it includes valuable information on which AST tree nodes each FermaT transformation is tested and possibly applied. Gathered information is also reused within

the presented transformation sequence prediction and elimination technique. All used FermaT transformations including their descriptions and applicability conditions can be found in Appendix A and A.5.

8.9 FermaT's Data Structures

As the FCE is equipped with a GUI it needs to be able to display and represent transformation processing steps. In order to do so, the illustrated FermaT transformation engine commands are utilised. To graphically represent the effects of transformations within WSL program sources, the (@Print WSL (@Program) " ") command is chosen, to guarantee that programs presented within the FCE are equivalent to the ones within the FermaT transformation engine.

The WSL programs represented within the engine are simply parsed by Java Tree methods. They collect all information needed to represent the engine AST within the FCE. To apply transformations on AST nodes, the above specified engine commands are illustrated to apply and navigate through the tree source. To assist direct jumps to AST nodes, Java methods are capable to generate FermaT tree paths. To generate a relationship between the represented AST and the internally loaded WSL program, the WSL syntax outlined in Appendix B.7 is adopted.

During the parsing process of AST, the FCE calculates for each node a specific AST tree node depth. This guarantees a proper relation between each AST node and the WSL program. The processed values are also used to limit the search depth of finding applicable transformations during the headnode's analysing process. Figure 8.2 presents the outcome of the "(@Print WSL (@Program))" engine command.

```
1  Statements
2     Cond
3     :  Guarded
4     :     Equal
5     :        Variable x__0
6     :        Number 0
7     :     Statements
8     :        Print
9     :        :  Expressions
10    :        :     String Goodby cruel world
11    :  Guarded
12    :     True
13    :     Statements
14    :        Print
15    :        :  Expressions
16    :        :     String Hello world
```

```
17    :  Guarded
18    :     True
19    :     Statements
20    :        Assignment
21    :        :  Assign
22    :        :     Var_Lvalue y__1
23    :        :     Number 2
24 #t
```

LISTING 8.3: FermaT transformation engine command: @Print_WSL (@Program)

8.10 Transformation Tasks Submission

The presented parallel transformations processing environment offers two job-submission techniques. Transformation tasks can be either submitted through the use of an OS terminal window or a client based FCE GUI . In both cases, the parallel transformation task related files need to be placed within a specified NFS folder. Once a transformation task is defined and submitted to the headnode, it will be automatically queued and parallelised based on its specification.

A command line sample to start the job-submission either in text ("-t") or in graphical ("-g") mode is listed below:

```
job.sh -t fc-head node3 example.ptd
```

- **job.sh:** Specifies the job-submission starting script used.
- **-t or -g:** Starts the job-submission either in text or graphical based mode.
- **fc-head:** DNS name of the headnode.
- **node3:** DNS name of the sender node.
- **example.ptd:** PTTD file.

8.11 Computing Node Processing

The computing node transformation processing steps are performed on a local basis, once the transformation task has been read and processing data has been remotely accessed. Computing nodes are equipped with a very lightweight FermaT transformation engine structure, designed to only receive transformation processing commands via the

described communication infrastructure. This light kernel serves the single purpose to compute transformation tasks in a fast and efficient way. In order to monitor parallel transformation processing behaviour of computing nodes, a GUI has been designed and implemented. This feature offers to monitor the parallel transformations processing behaviour of computing nodes. How the computing nodes are set-up to access processing data is illustrated within the separate available FCE tutorial.

8.12 FCE's Graphical User Interface

First it seemed to be the most appropriate way to extend the FermaT Maintenance Environment (FME) with parallel transformation processing functionalities. But after further analysis it was clear that this tool cannot fulfil the need for parallel transformations processing. The purpose of the FME is to serve as a demonstration tool. Changes would have been too huge and the interference with other researchers, working on this product-line, led to the solution of developing a standalone application. The FCE's GUI can be considered as the graphical front end for the proposed parallel transformations framework. The interface combines the outlined features and captures their behaviour visually. The system modules are grouped as:

- Communication Module.
- Computing Node Management Module.
- WSL Editor Module.
- Transformation Catalogue Module.
- Abstract Syntax Tree (AST) Module.
- Abstract Syntax Tree (AST) Node Module.
- Logging Module.

Figure 8.3 represents these modules. Compared to the ones used within the FME, these are redesigned as very lightweight and parallel processing adjusted modules.

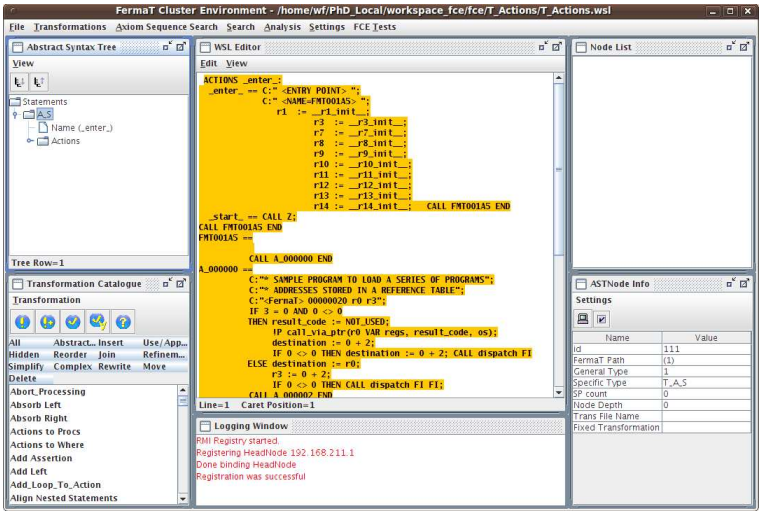


FIGURE 8.3: The Graphical User Interface (GUI) of the FCE

Communication Module:

As described, a communication system between the Fermat transformation engine and FCE was needed. The first part of the communication was established by the module developed for the FME. However its lack of performance needed to be solved. By tweaking and resolving these issues, the application time of a single FermaT transformation has been decreased to under 0.5 of a second. FermaT's transformation engine start files are indirectly executed when the FCE starts. Once the communication is established, the pipe-constructs handle the communication between the transformation engine and the FCE. All communication-and transformation-processes are solved by sending the illustrated FermaT specific commands across.

Computing Node Management Module:

Once a computing node registers its services within the headnode, its name is displayed within the Computing Node Management Module. Each computing node is displayed with its IP address and name. By right-clicking on a computing node item, it opens a drop-down facility to adjust hardware specific features or performance parameters. This information is either directly submitted by the computing node or formerly saved within the headnode database entries.

WSL Editor Module:

The WSL Editor Module serves as a one to one graphical presentation of the within the FermaT transformation engine loaded WSL program source. Changes to the WSL code result in a FermaT engine command. WSL code errors are automatically represented

within the AST view. The editor is very case sensitive and communicates directly with the FermaT engine, moving up and down the AST or loading new WSL program files.

Transformation Catalogue Module:

The Transformation Catalogue Module represents the WSL program transformations which are currently available within the FermaT transformation engine. All transformations are categorised based on their groups and sub-groups. WSL transformations can be tested and applied immediately on program code, based on their specification.

Abstract Syntax Tree (AST) Module:

The Abstract Syntax Tree (AST) Module is the graphical AST representation of the WSL program source loaded within the FermaT engine. The graphical tree can be used to navigate through the source or highlight WSL code sections. The AST module is mainly used to test and apply WSL program transformations and automatically alters once the program code is modified.

AST Node Module:

This module is generally used during the manual maintainer mode, in which the maintainer directly addresses the application of transformations to a specific computing node. The required information is temporarily stored within the headnode, before the specified assignments are submitted to the compute nodes.

Logging Module:

The Logging Module records all node behaviour, such as assignments by the headnode, submitted transformation task results, node failure or logging procedures. The information is also saved within a logging file for further analysis purposes.

8.13 FermaT Cluster Environment (FCE) Implementation

The implementation design of the FCE is very modular. Its core contains 20 classes. The whole FCE package includes over 200 classes, excluding jar archives and additional open-source libraries and components. FCE's main classes are listed below.

- **Framework Classes:**
  - **FCEMain:** The starting class of parallel transformations processing environment. All other classes are executed from this point on.
  - **FCEComponentregistry:** Manages all GUI related sub-components. They can be accessed via special parameter.
  - **ConnectionHandler:** Handles the communication of the system.
  - **RMIRegistry:** Manages all Java RMI based system activities.
  - **SocketRegistry:** Manages all TCP/IP based system activities.
  - **DatabaseManager:** Manages all database related tables and activities.
- **Configuration Classes:**
  - **FCEConfiguration:** Contains all FCE environment specific parameters such as FermaT console commands, NFS directory specifications and other static parameter which are accessed by the other system components.
- **Logging Classes:**
  - **FCELogger:** Logging system based on the Java Logging API.
  - **FCELoggerGUI:** Logs all parallel transformation processing activities performed by the headnode and computing nodes.
- **GUI Classes:**
  - **FCEGui:** Represents the main FCE GUI application and controls all graphical sub-components.
  - **Scheduling Table:** Displays the parallel transformation tasks, their computation time and their node assignment.
  - **Node Watch:** This utility can be used to access and control registered computing nodes remotely. It is mainly used to monitor transformation processes of computing nodes.

- **GUI Sub-Classes:**
  - **ASTTreeGUI:** Represents the AST of the WSL program source currently loaded within the FermaT transformation engine. The GUI can be utilised to navigate through the tree while collapsing and expanding it. Transformations can be "*pinned*" to AST nodes.
  - **AST Node Info:** Displays information of the within the AST module selected AST node: FermaT specific name, ID, possible and "*pinned*" transformations etc.
  - **Editor GUI:** Represents the currently loaded WSL program source.
  - **Computing NodeGUI:** Shows the within the FCE registered computing nodes. Listener implementations guarantee computing node configuration facilities.
  - **Transformation Catalogue GUI:** Represents and lists all WSL code transformations.
- **FermaT Specific Classes:**
  - **FCEConsole:** All FermaT specific procedures and commands are embedded within this class. This class establishes the connection pipes between the FermaT transformation engine and the FCE .
  - **FCEConsoleObserver:** Encapsulates all communication streams utilised to interact with the transformation engine. This class represents the closest layer to the transformation engine.
- **Visualisation Components:**
  - **WSL Program Dependency Graph:** Visualises the dependency between the WSL AST types of the program source as a graph.
  - **Cluster Graph:** Presents the parallel environment graphically, this includes headnode and computing nodes.
- **Computing Node package:**
  - **ClientConsole:** A lightweight client communication console. It embeds all FermaT specific procedures and represents the main connection between the computing node and the client based transformation engine.
  - **ClientConsoleObserver:** Encapsulates all communication streams utilised to interact with the client transformation engine. This class represents the closest layer to the FermaT specification.

8.14 Summary

This chapter gave an insight and review of the prototype tool developed for the proposed parallel transformations framework. It illustrated its concept, features and the implementation needed to establish a parallel transformations processing system. It demonstrated how the system can be accessed and which components are mandatory to satisfy the presented work. FermaT specific parallel transformations processing steps were outlined, and their structures were discussed. At the end the FCE's Graphical User Interface (GUI) are illustrated and their implementation is reviewed.

Chapter 9

Case Studies

Objectives

- To present and describe the use of the proposed parallel transformations framework on the basis of several examples.
- To illustrate the difference within the presented parallel transformations processing techniques.
- To demonstrate and evaluate the practical need for the presented research.

9.1 Introduction

This chapter presents two case studies to illustrate the practical need of the presented research. Within both, transformation scheme descriptions are decomposed parallelised and their processing time and results are evaluated. The first sample transformation scheme description uses two abstraction level constraints and focuses on raising the overall abstraction level of a WSL program. The second one is more complex, utilises six constraints with the emphasis on raising the overall execution speed of a program. The case studies main intention is to demonstrate how parallel processing can be utilised to automate and speed-up transformation tasks. In correspondence to this, the overall parallel processing constraint for these two case studies is aimed on the decrease of the overall computing time. It has to be remembered, that the specified parallel processing

constraints "C_p" do not affect in any way the transformation scheme descriptions embedded within reengineering constraints "C_r". In addition, both case studies are evaluated on the basis of the proposed parallel transformations processing techniques, while their advantages and weaknesses are presented.

9.2 Parallel Transformations Processing Environment

In order to present a parallel processing solution within the domain of program transformation application and the FermaT transformation system, a research specific parallel transformations processing environment has been established. Its architecture is outlined in detail in Chapter 8. The advance of computing transformations in parallel is going to be demonstrated through the following architectural design; a cluster based architecture and consisting of Commercial Off-The-Shelf (COTS) components, with one headnode and six computing nodes. The system has the following hardware and software specific setup:

1 Headnode:
CPU: Intel Pentium IV 2,66 GHz
RAM: 1 GB
HDD: 60 GB
Linux OS: Ubuntu 8.10 Desktop
Linux Kernel Version: 2.6.27
FermaT Version: *fermat3* (Ver. July 2009)
Java Version: 1.6
Perl Version: 5.8.8
PostgreSQL Version: 8.3.9

6 Computing Nodes:
CPU: Intel Pentium III 933 MHz
RAM: 256 MB
HDD: 20 GB
Linux OS: Ubuntu Server 8.10
Linux Kernel Version: 2.6.27
FermaT Version: *fermat3* (Ver. July 2009)
Java Version: 1.6
Perl Version: 5.8.8

The environments parallel computing power can be easily expanded, by dynamically adding more computing resources to the system during runtime. This potential is described in Chapter 8 Tool Support and within a separate available FCE tutorial.

9.3 Headnode Analysis

The system analysis performed by the headnode plays a major role in the process of parallel transformations computing. To recap, the headnode performs the following preprocessing steps before parallel tasks are submitted to the computing nodes for parallel computation:

- Analysis of the parallel computing environment.
- Evaluation of the parallel transformation task description.
- Analysis of the WSL program source file and its AST.
- Transformation scheme description and reengineering constraint investigation.
- Decomposition of the specified transformation scheme description on the basis of parallel processing constraints.
- Evaluation of fulfilment of specified parallel processing constraints.
- Parallel transformation task submission and result evaluation.

The following sections outline these headnode and computing node specific processes on the basis of the presented case studies in more detail.

9.4 Environment Analysis

As mentioned, the cluster environment analysis is one of the key aspects to evaluate, to decide if specified parallel processing constraints can be fulfilled or not. In relation to this, the parallel processing environment analysis plays an important role for the speed-up of parallel transformation tasks. As has been outlined in Chapter 5.2, all work-nodes have to register their services within the headnode. Furthermore, each computing node has to undergo a performance test to evaluate its transformation computing power. As all computing nodes within these studies are equipped with the same hardware and software, they are judged with the same transformation processing speed. Therefore the load-balancing concept is equally treated. On the basis of these values, suitable parallel transformations processing roadmaps are generated.

9.5 Computing Node Performance Tests

Within the next processing step, an environment analysis on the basis of the FermaT transformation performance test has to be evaluated. Additional information about the communication between the headnode and its computing nodes has to be consolidated. Running these evaluations produces the following data:

- **FermaT Transformations Test:** Overall Test Time: 118 ms.
- **RMI Performance Test:** 5000 Remote Calls: 5779 ms (5,79 s).
- **TCP/IP Performance Test:** 5000 Remote Calls: 4980 ms (4,98 s).

Analysis of this FermaT transformation test reveals that the average transformation processing time is close to 1ms a transformation. The evaluation of computing node calls over the RMI or TCP/IP stack shows that the communication delay is close to 1ms a call. Having gathered this information, processing time for each transformation step can be closely predicted. This information will be further processed by the headnode. More information about the test can be found in Appendix F.2.

9.6 Case Study 1

In general, parallel computing techniques can help reduce the time it takes to reach a solution. To derive the full potential of parallelism, it is important to choose an approach which is appropriate for the optimisation problem. Within this case study two parallel transformations processing techniques, "*parallel processing*" and "*linear line computing*", are demonstrated and their effects are discussed. To lead to a successful parallel processing and reengineering solution, the definition of constraints is mandatory. In regard to this, the following parallel processing and reengineering constraints have been defined:

- The "*overall parallel processing constraint*" for this particular case study is speed-up. Further parallel transformation task refinement results in parallel processing and linear line computing constraints according to the formal language specified and described in Chapter 4.6. The first results in a simple Divide and Conquer algorithm, whereas the second technique is based on the linear line processing concept. Both have been outlined in Chapter 7.

- The "*overall reengineering constraint*" in this context is the achievement of a higher-level of abstraction of the WSL program listed in Listing 9.6.1. This can only be achieved by the application of program transformations, which are specified within the outlined transformation scheme description.

It has to be remembered, that the parallelising constraints "C_{pn}" defined, do not effect in any way the generation and evaluation of transformation sequences and constraints introduced to lead to the desired reengineering aim.

9.6.1 Program Analysis and Transformation Scheme Description Definition

Program analysis in this context has to do with the definition of the applicability and application of program transformations. During a software reengineering process usually many thousands of program transformations have to be applied. Knowledge of applying these transformations in the right order to successfully lead to the desired reengineering aim is the challenging task. To assist the maintainer by defining such a task transformation scheme descriptions are utilised. Within a transformation scheme description the success of a transformation process can be considered as a combination of constraints and transformation satisfaction. These maintenance goals can be reached in an automated manner by the utilisation of the proposed parallel transformation processing environment.

The "*overall maintenance goal*" of the WSL program presented in Listing 9.1 is the rise to a high-level of abstraction. In order to fulfil this objective, the analysis of the program source is the main need. Nevertheless, the maintainer also has to have a little knowledge which transformations need to be chosen and in which order they need to be applied, to be able to achieve the specified reengineering aim. The combination of the usage of FermaT transformations and the utilisation of transformation scheme descriptions assist the maintainer with this achievement. Analysing the specified WSL program reveals the following structure:

1. It has an "*Action System*", which is a low-level construct according to the WSL specification.
2. It consists of "four *Actions*": Prog, A, B, C of which one is recursive ("*Action B*", marked green).

```
ACTIONS PROG:
  PROG ==
    i := 55;
    j := k;
    IF k < 0 THEN
      j := j * 5;
      CALL C
    ELSIF k < 25 THEN
      j := j * 2;
      CALL C
    FI;
    CALL A
  END
  A ==
    IF i > j THEN
      k := k + i;
      CALL C
    FI;
    CALL B
  END
  B ==
    IF j < 75 THEN
      j := j + 5;
      k := k * 2;
      CALL B
    FI;
    CALL C
  END
  C ==
    i := 0;
    CALL Z
  END
ENDACTIONS
```

Listing 9.1: Case Study 1 WSL Program

In the perspective of the maintainer and the desired reengineering aims, the circumstances followed like to be resolved through the application of the specified program transformations cited in Listing 9.2:

- Achievement of a higher-level of abstraction of the WSL program, by eliminating the "*Action System*" (WSL AST type "*T_A_S*").
- Elemination of the "*Action System*" through a transformation of the recursive "*Action B*" into a "*DO-loop*".

The second reengineering aim which is the elimination of the "*DO-loop*" (AST type "*T_Floop*"), can be resolved through the introduction of a "*While-loop*" (AST type

"*T_While*"). Having defined these reengineering aims, the following transformations have been chosen to be the most appropriate ones:

- **Remove Recursion in Action**: Removes the recursion within "*Action B*".
- **Substitute and Delete:** Simplifies the program code.
- **Simplify Item:** Simplifies the "*Action System*", expressed by the WSL AST type ("*T_ A_S*").
- **Floop to While:** Transforms the "*DO loop*" into a "*While loop*". The "*DO loop*" was introduced by the transformation "*Remove Recursion in Action*" which removed the recursion in "*Action B*".

According to the specification of the formal language to describe transformation scheme descriptions, the following description is outlined to achieve the desired reengineering aims.

```
1 (
2   (
3     < Remove Recursion in Action > |
4     < Substitute and Delete > |
5     < Simplify Item @ T_A_S > |
6     < Floop to While >
7   ) [1 .. 6]
8 ) {C1, C2}
```

LISTING 9.2: Case Study 1 Transformation Scheme Description

To support the effectiveness of this transformation scheme description the maintainer has chosen an alternative-construct to be the most suitable solution for the specified reengineering task. In addition, the range of this technique has been increased by adding a quantifier construct "[1..6]" to its definition. This results in a maximum transformation sequence length of six transformations. The number "6" has been chosen due to the following facts:

1. **Step:** Removing the recursion (1 transformation).
2. **Step:** Simplifying the 3 Actions (3 transformations).
3. **Step:** Removing the Action System (1 transformation).
4. **Step:** Removing the DO-loop (1 transformation).

To fulfil the desired reengineering aim, two constraints are added to the transformation scheme description alternative-construct. Their fulfilment is checked each time a program transformation is applied, as discussed in Chapter 7. The two reengineering constraints are specified as:

- **C1:** A high-level constraint, which specifies a group of AST types, should ensure that the specific AST type "*T_A_S*" which stands for an "*Action System*" and a low-level WSL construct, should not occur within the final state of the program "P_i".
- **C2:** The second constraint is a conventional constraint, and states that a "*DO-loop*" which is usually introduced to eliminate a recursion should not occur within the final state of any transformed program "P_i". More precise, the AST specific type "*T_Floop*" should therefore not appear within any program final state "P_i".

9.6.2 Transformation Scheme Description Decomposition

Within the next processing step, the headnode has to analysis the specified parallel transformation task and its parallel computing constraints. Based on its evaluation, the headnode calculates a suitable parallel transformations processing outline. The environment analysis plays a major role, as transformation scheme descriptions are usually decomposed based on the number of available computing nodes. According to the definition of laws to decompose transformation scheme descriptions, specified in Chapter 6, the outlined transformation scheme descriptions are decomposed. To comprehend the decomposition procedure within this case study, the utilised transformations are substituted based on the Table 9.1 below.

Transformations	Acronym
< Remove Recursion in Action >	T_0
< Substitute and Delete >	T_1
< Simplify Item @ T_A_S >	T_2
< Floop to While >	T_3

Table 9.1: Substitution of Transformation Scheme Description: Case Study 1

With the table above the transformation scheme description in Listing 9.2 can be specified as:

$((T_0 \mid T_1 \mid T_2 \mid T_3)[1 \,..\, 6]\,)\{\text{C1, C2}\}$

On the basis of the laws specified for an alternative with an additional quantifier construct "[1..6]", the embedded alternative-construct can be decomposed into the following sub-schemes:

Iteration (1): $(T_0 \mid T_1 \mid T_2 \mid T_3)$
Iteration (2): $(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$
Iteration (3): $(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$
Iteration (4): $(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$,
$(T_0 \mid T_1 \mid T_2 \mid T_3)$
Iteration (5): $(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$,
$(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$
Iteration (6): $(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$,
$(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$, $(T_0 \mid T_1 \mid T_2 \mid T_3)$

Further decomposition of the iteration construct would produce 5460 transformation sequences. As the decomposition procedure starts, the first transformation sequences are:

Iteration (1): $T_0 \parallel T_1 \parallel T_2 \parallel T_3$
Iteration (2): $(T_0, T_0) \parallel (T_0, T_1) \parallel (T_0, T_2) \parallel (T_0, T_3) \parallel$
$(T_1, T_0) \parallel (T_1, T_1) \parallel (T_1, T_2) \parallel (T_1, T_3) \parallel$
$(T_2, T_0) \parallel (T_2, T_1) \parallel (T_2, T_2) \parallel (T_2, T_3) \parallel$
$(T_3, T_0) \parallel (T_3, T_1) \parallel (T_3, T_2) \parallel (T_3, T_3)$
Iteration (3): $(T_0, T_0, T_0) \parallel (T_1, T_0, T_0) \parallel (T_0, T_1, T_0) \parallel (T_0, T_0, T_1) \parallel$
$(T_1, T_1, T_0) \parallel (T_1, T_0, T_1) \parallel (T_0, T_1, T_1) \parallel (T_1, T_1, T_1)$
....

until the end:

Sequence 5457: $(T_3, T_3, T_3, T_3, T_3, T_0)$ C1,C2
Sequence 5458: $(T_3, T_3, T_3, T_3, T_3, T_1)$ C1,C2
Sequence 5459: $(T_3, T_3, T_3, T_3, T_3, T_2)$ C1,C2
Sequence 5460: $(T_3, T_3, T_3, T_3, T_3, T_3)$ C1,C2

Processing all transformation sequences within a single processing environment would result in a tremendous computing task of applying nearly over 30,000 program transformations. This would result in a computing time of nearly 8 hours. This computing time can be decreased by utilising the parallel transformations processing techniques proposed within this thesis. Further analysis of the generated transformation sequences

in combination with the specified WSL program analysis reveals that over 30 % of the generated transformation sequences can be removed. The basis for this knowledge is illustrated in the Table 9.2 below.

Transformation	Applicable on: AST Type	Introduced Types
T_0	T_Action	T_Floop
T_1	T_Action, T_Funct, T_Proc	
T_2	T_A_S	
T_3	T_Floop	T_While

TABLE 9.2: Program Transformation Effects

Evaluating the presented data reveals that the transformation "T_0" introduces the AST type "*T_Floop*" within the specified WSL program. Knowing that transformation "T_3" is only applicable on the specific AST type "*T_Floop*", specifies that transformation sequences which start with "T_3" before transformation "T_0" is stated, can be totally removed. Furthermore, all transformation sequences in which "T_0" is not cited can be removed, since one of the reengineering constraints is that the Action System should be removed from the final program state "P_i". This cannot be fulfilled since only transformation "T_0" makes this possible by eliminating the recursion in "*Action B*". Furthermore, within each valid transformation sequence, transformation "T_3" has to occur after "T_0", since this transformation produces the "*While-loop*" which satisfies constraints "C_2". It has to be stated, that only both high-level and conventional constraints specified within the transformation task have to be fulfilled to be "*true*", to satisfy the final WSL program state "P_i".

Based on the elimination of unsatisfied transformation sequences, the remaining transformation sequences are grouped into packages of independent sub-schemes. These schemes can be run independently in parallel. Listing 9.3 shows this kind of grouping for the presented case study and the assignment to 3 computing nodes. As explained in Chapter 6, the alternative-construct serves as an ideal starting point to evaluate parallelism. Further examination of these sub-schemes is now undertaken, as the specified environment is equipped with 6 computing nodes.

```
1 (<T0>,(<T0>|<T1>|<T2>|<T3>)[0..5]){C1,C2}
2 (<T1>,(<T0>|<T1>|<T2>|<T3>)[0..5]){C1,C2}
3 (<T2>,(<T0>|<T1>|<T2>|<T3>)[0..5]){C1,C2}
```

LISTING 9.3: 3 Transformation Sub-Schemes of Case Study 1

The next decomposition step produces 6 sub-schemes by evaluating the alternative-construct shown in Listing 9.4. Parallel headnode analysis also reveals that two sub-schemes embed transformation "T_3" before "T_0" is applied.

```
1 (<T0>,((<T0>|<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
2 (<T0>,((<T2>|<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
3 (<T1>,((<T0>|<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
4 (<T1>,((<T2>|<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
5 (<T2>,((<T0>|<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
6 (<T2>,((<T2>|<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
```

Listing 9.4: 6 Transformation Sub-Schemes of Case Study 1

Further decomposition of the presented sub-schemes eliminates unsatisfactory sub-schemes, demonstrated in Listing 9.5.

```
1  (<T0>,(<T0>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
2  (<T0>,(<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
3  (<T0>,(<T2>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
4  (<T0>,(<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
5  (<T1>,(<T0>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
6  (<T1>,(<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
7  (<T1>,(<T2>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
8  (<T2>,(<T0>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
9  (<T2>,(<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
10 (<T2>,(<T2>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
11
12 ....
13 (<T1>,(<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
14 (<T2>,(<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
```

Listing 9.5: 10 Transformation Sub-Schemes of Case Study 1

As the decomposition technique illustrates there will be not always an even number of transformation sub-schemes matching the number of computing nodes. Usually the headnode would stop to produce more transformation sub-schemes than there are computing nodes available. In this case, the headnode would stop at a total number of 6 sub-schemes.

Based on the gathered information of decomposition in combination with the availability of computing nodes, the headnode can estimate the overall computing time by dividing the left-over sub-schemes by the number of computing nodes. During this estimation process, the developed computing node performance test plays a major role, knowing how long each individual transformation sequence takes to be performed. Although it has to be kept in mind that this process depends a lot on the definition of the transformation scheme description and the citation of the referred AST path of each FermaT

transformation. As already demonstrated in Chapter 7.2.1, that can be expressed in a very dynamic manner. Judging this information, the headnode is able to evaluate if the parallel processing constraints can be fulfilled before the parallel process starts. The basis for this knowledge is the developed WSL transformation analysis table presented in Appendix A.5.

9.6.3 Parallel Transformation Processing Techniques

On the basis of the generated information, sub-schemes and specified parallel constraints, a suitable parallel transformation processing outline is evaluated. In perspective to these case studies, the following sub-sections present examples of both parallel techniques specified.

9.6.4 Parallel Transformation Processing Case Study 1

Based on the number of computing nodes within the parallel processing environment, the evaluated sub-schemes can be further decomposed. However the quantifier construct within the original transformation scheme description specifies its limitation. It has to be noted, the more precise sub-schemes are defined, the easier the elimination technique can eliminate inapplicable transformation sequences. The presented technique discovers unreachable transformation sequences more quickly and therefore redundant work can be avoided. Listing 9.6 presents the transformation sub-schemes which are assigned to the specified parallel transformation processing environment.

```
1 (<T0>,((<T0>|<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
2 (<T0>,((<T2>|<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
3 (<T1>,((<T0>|<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
4 (<T1>,((<T2>|<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
5 (<T2>,((<T0>|<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
6 (<T2>,((<T2>|<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
```

Listing 9.6: 6 Transformation Sub-Schemes of Case Study 1

Knowledge gathered by the headnode, knowing that the specified cluster environment consists of a maximum number of 6 computing nodes, stops this analysing process and saves the generated sub-schemes within separate files. The file names consist of a combination of the original transformation scheme name and the transformation task sub-ID.

9.6.5 Parallel Transformation Task Generation Case Study 1

The next processing step results in the construction of the "*PLACED PAR*" constructs, utilised to assign the generated transformation tasks to the computing nodes. Their specification by utilising the TCP/IP communication layer is presented in Listing 9.7. The next processing step would result in the acknowledgement by the computing nodes that they start their parallel transformation processing. It ends in collecting the assigned transformation task information gathered from the NFS repository.

```
1 PLACED PAR
2   fc-node1 TCP/IP (1, caseStudy1.wsl, task1.tdsl)
3   fc-node2 TCP/IP (2, caseStudy1.wsl, task2.tdsl)
4   fc-node3 TCP/IP (3, caseStudy1.wsl, task3.tdsl)
5   fc-node4 TCP/IP (4, caseStudy1.wsl, task4.tdsl)
6   fc-node5 TCP/IP (5, caseStudy1.wsl, task5.tdsl)
7   fc-node6 TCP/IP (6, caseStudy1.wsl, task6.tdsl)
```

Listing 9.7: PLACED PAR Case Study 1

9.6.6 Parallel Processing Results Case Study 1

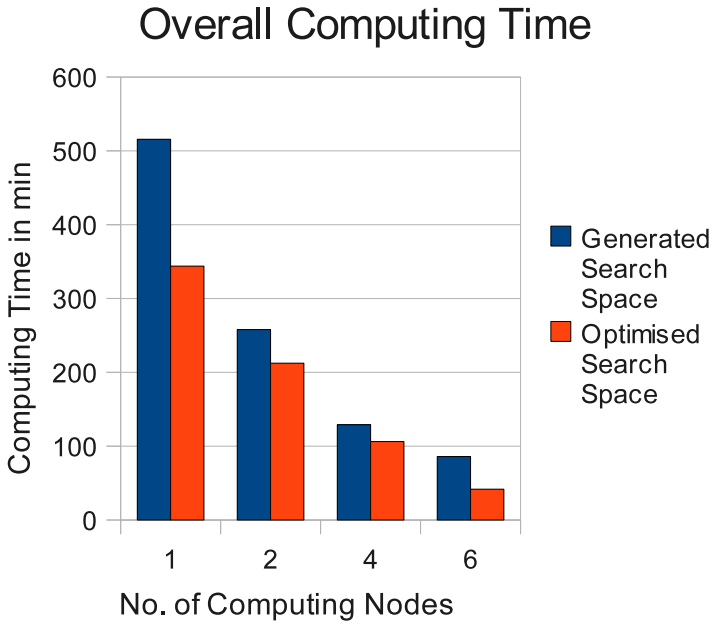


Figure 9.1: Overall Computing Time: Case Study 1

Overall Parallel Computing Time in Minutes				
	Number of Computing Nodes			
	1	**2**	**4**	**6**
Generated Search Space	515.8	257.9	128.95	85.97
Optimised Search Space	343.87	212.4	106.22	41.73
Time saved by optimisation	171.93	45.25	22.73	44.23

TABLE 9.3: Overall Computing Time: Case Study 1

With the objectives for speed-up and fulfilment of the specified parallel transformation task, it can be concluded according to the charts and tables, that by sub-dividing the transformation scheme description into independent sub-schemes enormous speed- up can be achieved. This is not only due to the fact that the elimination of unsatisfied transformation sequences are removed beforehand. With the assistance of grouping transformation sequence, redundant transformation work is avoided and tremendous computing time is saved compared to the normal parallel task submission. In concern of the overall fulfilment of the defined reengineering aim, the transformation sequence "(<T0>,<T1 >,<T1>,<T1>,<T2>,<T3>) {C1, C2}" leads to the satisfaction of the specified reengineering constraints "C_1" and "C_2". This result is evaluated in a processing time of 32.5 min by the assigned computing node 1.

9.6.7 Parallel Linear Array Processing Case Study 1

Parallel linear line processing is another concept of parallel transformations processing which has been discussed in closer detail in Chapter 7.4.2. This section illustrates and outlines this technique in regard of this case study. The decomposition of transformation scheme descriptions according to linear line processing is performed in a similar pattern to the one specified above. The only difference is that the transformations or sequence of transformations and alternative-constructs of individual sub-schemes are assigned to individual computing nodes. Grouped, they function as a linear line of processing computing nodes and only exchange transformed WSL program files. The main purpose of utilising the linear-line construction is to process more than one WSL program at a time. In the case of these case studies this parallel transformations processing technique should illustrate its capabilities. Unrolling and grouping the generated sub-schemes of this case study, according to the specified decomposition techniques, is presented in Listing 9.8 below.

```
1 (<T0>,((<T0>|<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
2 (<T0>,((<T2>|<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
3 (<T1>,((<T0>|<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
4 (<T1>,((<T2>|<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
5 (<T2>,((<T0>|<T1>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
6 (<T2>,((<T2>|<T3>),(<T0>|<T1>|<T2>|<T3>)[0..4])[0..1]){C1,C2}
```

LISTING 9.8: 6 Transformation Sub-Schemes of Case Study 1

The technique already reveals its capabilities by demonstrating that sub-scheme elimination depends on the decomposition and grouping of particular transformation sequences, explained in Chapter 6.12.

A by-product of this technique is that all generated sub-schemes have the same length, so therefore the overall computing time of the computing nodes estimated by the headnode can be equally treated. Table 9.6.7 presents the assignment of the 6 sub-schemes to 6 linear-computing lines.

Sub-Scheme Assignment			
	Computing Node		
	1	2	3
1	(T0) {C1,C2}	(T0 | T1)[0..1] {C1,C2}	(T0 | T1 | T2 | T3)[0..4] {C1,C2}
2	(T0) {C1,C2}	(T2 | T3)[0..1] {C1,C2}	(T0 | T1 | T2 | T3)[0..4] {C1,C2}
3	(T1) {C1,C2}	(T0 | T1)[0..1] {C1,C2}	(T0 | T1 | T2 | T3)[0..4] {C1,C2}
4	(T1) {C1,C2}	(T2 | T3)[0..1] {C1,C2}	(T0 | T1 | T2 | T3)[0..4] {C1,C2}
5	(T2) {C1,C2}	(T0 | T1)[0..1] {C1,C2}	(T0 | T1 | T2 | T3)[0..4] {C1,C2}
6	(T2) {C1,C2}	(T2 | T3)[0..1] {C1,C2}	(T0 | T1 | T2 | T3)[0..4] {C1,C2}

TABLE 9.4: Linear Array Sub-Scheme Assigning: Case Study 1

Once this evaluation is performed, each of the above sub-schemes are assigned to the proposed parallel processing architecture. However as the number of computing nodes within this environment is limited to six, only two sub-schemes at a time can be run in parallel. Because of the above, this already demonstrates that 3 computing nodes are needed to run one sub- scheme. On the other side, this table already presents, not only the number of computing nodes within a sub-scheme definition which play a major role, but also that the total number of independent sub-schemes is more important to attain the achievable speed-up.

9.6.8 Results and Summary Case Study 1

On the basis of the number of available computing nodes within the system and the chosen parallel transformations processing architecture illustrated in the presented case study, the transformation sequence shown in Listing 9.9 satisfies **all** given reengineering constraints. As already outlined in the introductory section, the recursion within "*Action B*" had to be removed by the application of the "*Remove Recursion in Action*" transformation. The next transformation processing steps included the application of three times the transformation "*Substitute and Delete*" to substitute and remove the other WSL language specific actions within the program code "P_1". Since other reengineering constraints has been specified by converting the "*DO-loop*" to the "*While-loop*", this action has been performed by the "*Floop to While*" transformation. This results in the application of the transformation "*Simplify Item*" on the WSL specific AST type "*T_A_S*".

```
1
2  < Remove Recursion in Action @ /0,1,2/ > ,
3  < Substitute and Delete @ /0,1,1/ > ,
4  < Substitute and Delete @ /0,1,1/ > ,
5  < Substitute and Delete @ /0,1,1/ > ,
6  < Floop to While @ /2,2,1,0,1,1,0/ >
7  < Simplify Item @ /0/ > ,
```

Listing 9.9: Case Study 1 Satisfying Transformation Sequence

Once the above transformation processing steps are performed, the resulting WSL code presented in Listing 9.10, fulfils the stated reengineering constraints "C_0" and "C_1".

```
1  i := 55;
2  j := k;
3  IF k < 0
4    THEN j := j * 5; i := 0
5  ELSIF k < 25
6    THEN j := j * 2; i := 0
7    ELSE IF i > j
8           THEN k := k + i; i := 0
9           ELSE DO IF j < 75 THEN j := j + 5; k := k * 2 ELSE EXIT(1) FI
10               OD;
11               i := 0 FI FI
```

Listing 9.10: Case Study 1 Result: WSL Program P_n

To conclude the matter of parallel transformation processing and the achievement of the satisfaction of the presented reengineering goals, Figure 9.2 presents the comparison of

both parallel processing techniques described. Briefly referring to the Chart 9.2 as well as the Table 9.5 below, both state and confirm that the division of the transformation scheme description into individual sub-schemes contribute the most to the computing time saving factor. These results were only achievable by the introduction of the presented analysing system, which evaluates and eliminates inapplicable and redundant transformation sequences beforehand.

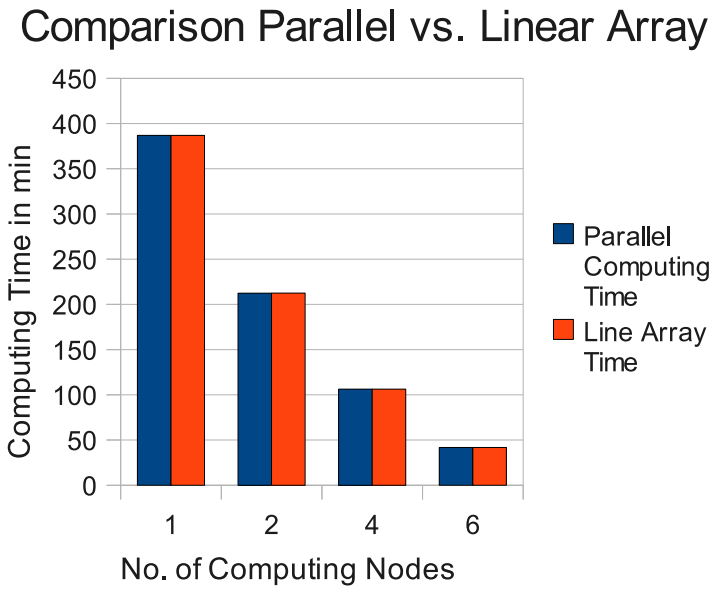


FIGURE 9.2: Comparison of the Overall Computing Time: Case Study 1

Overall Parallel Computing Time in Minutes				
	Number of Computing Nodes			
	1	**2**	**4**	**6**
Generated Search Space	515.8	257.9	128.95	85.97
Parallel Computing	386.85	212.4	106.22	41.73
Linear Line Computing	386.85	212.43	106.22	41.78
Time saved by optimisation	171.93	45.25	22.73	44.23

TABLE 9.5: Comparison Overall Computing Time: Case Study 1

Focusing on the overall performance compared to a single processing environment in Table 9.5 above, demonstrates the tremendous parallel computing power, and the parallel effectiveness which can be achieved by unrolling the transformation scheme description into independent transformation sequences and grouping them into efficient sub-schemes. The network communication only plays a minor role in this case study because only two transformed WSL program files are passed at the beginning of the parallel transformation

process between computing node “2 and 3”, illustrated in Table 9.6.7. Additionally, the stated results also reproduce the difference and effects of both parallel transformations processing techniques presented. Nonetheless the linear processing type only reveals its true potential by computing more than one WSL program file at the same time, trying to fulfil a precisely defined reengineering goal.

9.7 Case Study 2

The overall objective of this case study is to demonstrate that more complex and time consuming transformation tasks can be solved in a reasonable length of time. The first case study dealt with a total of 5460 transformation sequences, of which more than 30 % are removed by the presented analysing technique utilised by the headnode.

This example deals with a total number of 37400 transformation sequences. A single processing PC would need 22 hours to compute the generated transformation sequence search-space. In comparison to the first case study, an appropriate parallel computing solution has to be found in reasonable time. Similar to the first, the presented parallel transformation processing techniques, parallel transformations processing and linear-line transformations processing are evaluated and their results and effects are discussed.

In regard to the lead to a successful parallel processing and reengineering solution, the definition of constraints is mandatory. The following parallel processing and reengineering constraints have been specified for this case study. Similar to the ones in case study 1:

- The "*overall parallel processing constraint*" for the second case study is speed-up. Further refinement of the parallel processing constraints would result in the definition of parallel transformations processes. The two techniques are based on the "*parallel-* and *linear-line- processing*" concept.
- The "*overall reengineering constraint*" for the presented study is the increase of execution speed of the presented WSL program.

Given those constraints the following sections will outline the parallel transformation processing steps according to the defined aims.

9.7.1 Program Analysis and Transformation Scheme Description Definition

Similar to the program specified in the first case study, the presented WSL program of this case study also serves as a demonstration. In this context, the description within the transformation scheme of embedded reengineering constraints belongs to the group of behaviour constraints outlined in Chapter 3.4. Each specified constraint is based on software metric constraints which in this case include numerical values to be satisfied. The program which is classified for this demonstration example is presented in Appendix E.

The intention of the WSL program source is to fill a one-dimensional array with integer values and sort them during runtime. Starting from the lowest to the highest values, utilising the "*bubblesort algorithm*". In addition, the program also embeds a procedure which accesses the array via a binary search. The main reengineering constraint is to increase the execution speed of the presented program. As execution speed is usually evaluated during program execution, which cannot be evaluated during a transformation process. The focus of this case study lies on WSL program analysis to speed-up the program execution through program restructuring.

Therefore during each program transformation application, the following low-level behavioural constraints are proved. They further embed 4 execution speed constraints and guides to improve the overall execution speed. These constraints are needed for verification and evaluation purposes after the application of each program transformation:

- **Constraint** C_I focuses on a local program property of the execution speed of *an addition* The constraints is satisfied if the specific AST type "*_Plus* " does not occur in any WSL program state "P_i".
- **Constraint** C_{II} focuses on a local program property of the execution speed of *a subtraction* The constraints is satisfied if the specific AST type "*T_Minus*" does not occur in any WSL program state "P_i".
- **Constraint** C_{III} focused local program property of the execution speed of *a multiplication* The constraints is satisfied if the specific AST type "*T_Times*" does not occur in any WSL program state "P_i".
- **Constraint** C_{IV} focused local program property of the execution speed of *a division* The constraints is satisfied if the specific AST type "*T_Div*" does not occur in any WSL program state "P_i".
- **Constraint** C_V focused local program property of the execution speed of *an assignment* The constraints is satisfied if the specific AST type "*T_Assign*" does not occur in any WSL program state "P_i".
- **Constraint** C_{VI} focused local program property of the execution speed of *a procedure call* The constraints is satisfied if the specific AST type "*T_Proc_Call*" does not occur in any WSL program state "P_i".

The main assumption and reengineering purpose of this case study is the fulfilment of the presented constraints. This is achieved by evaluating each WSL program state result "P_i", if it is satisfied or not. In more concrete terms, the number of AST types after each FermaT transformation has to be evaluated and checked against the embedded metric

constraints, as a matter of execution speed. Metric constraints determine the maximum number of unsatisfied execution speed constraints within a particular program state WSL program "P_i". In terms of this case study, there are four metric constraints "$C_1 ... C_4$" involved within each program transformation process. These constraints are directly included within the specified transformation scheme description as:

- **Constraint** C_1 states that a program state should contain less than 13 dissatisfied "C_I" or "C_{II}" constraints. More precisely it is satisfied, if less than 13 AST specific types of "*T_Addition*" or "*T_Subtraction*" are included within any program state.
  .
- **Constraint** C_2 states that a program state should contain less than 4 dissatisfied "C_{III}" or "C_{IV}" constraints. More precisely it is satisfied, if less than 4 AST specific types of "*T_Times*" or "*T_Div*" are included within any program state.
- **Constraint** C_3 states that a program state should contain less than 25 dissatisfied "C_V" constraints. More precisely it is satisfied, if less than 25 AST specific types of "*T_Assign*" are included within any program state.
- **Constraint** C_4 states that a program state should contains no dissatisfied "C_{VI}" constraints. More precisely it is satisfied, if no AST specific types of "*T_Proc_Call*" are included within any program state.

This leads to a transformation scheme description to support the fulfilment and satisfaction of 4 overall constraints. The search tactics for their satisfaction is important and focuses on basic steps of:

- The use of transformation "*Merge Right*" to merge the program array switching.
- Part two tries to reduce the number of variables. An alternative construct is utilised for this purpose and seems to be the most appropriate solution, by simplifying the WSL statements.
- The program needs to be simplified at the end.
- The satisfaction of constraint four by the removal of the procedure call, AST type "*T_Proc*".

With this evaluated information, Listing 9.11 presents a transformation scheme description of:

```
(
  < Merge Right @ /0,1,2,3,6,1,0/ >,
  < Merge Right @ /0,1,2,3,6,1,0/ >,
  (
    (
      < Remove All Redundant Variables @ // > |
      < Constant Propagation @ // > |
      < Delete Unreachable Code @ // > |
      < Delete All Redundant @ // >
    ) [1 .. 4],
    < Simplify @ // > [0 .. 1],
    < Simplify Item @ T_Assign > [0 .. 10]
  ) {C1, C2, C3},
  < Substitute and Delete @ T_Proc > [1 .. 5]
) {C4}
```

LISTING 9.11: Case Study 2 Transformation Scheme

9.7.2 Transformation Scheme Description Decomposition

Following the same procedures as in case study 1, the transformation scheme description has to be analysed and decomposed to run in parallel. Based on the laws of decomposition, the transformation scheme description is decomposed. To more easily comprehend the decomposition technique, the transformations which are used within the transformation scheme description are substituted according to the Table 9.6 below.

Transformation	Acronym
< Merge Right >	T_0
< Remove All Redundant Variables @ // >	T_1
< Constant Propagation @ //>	T_2
< Delete Unreachable Code @ // >	T_3
< Delete All Redundant @ // >	T_4
< Simplify @ // >	T_5
< Simplify Item @ T_Assign >	T_6
< Substitute and Delete @ T_Proc >	T_7

TABLE 9.6: Substitution Transformation Scheme Case Study 2

Given the table above and the definition of laws described in Chapter 6, the transformation scheme description presented in Listing 9.11 can be substituted to:

$(T_0, T_0, ((T_1 \mid T_2 \mid T_3 \mid T_4)[1..4]), T_5[0..1], T_6[0..10])\{C_1,C_2,C_3\}, T_7[1..5])\{C_4\}$

The next processing step results in the identification of alternative-constructs within the presented scheme. As mentioned, this construct serves as an ideal way to achieve

parallelism. After its identification: $((T_1 \mid T_2 \mid T_3 \mid T_4)[1..4])$, it is decomposed into the following procedures:

Iteration (1): $(T_1 \mid T_2 \mid T_3 \mid T_4)$
Iteration (2): $(T_1 \mid T_2 \mid T_3 \mid T_4)$, $(T_1 \mid T_2 \mid T_3 \mid T_4)$
Iteration (3): $(T_1 \mid T_2 \mid T_3 \mid T_4)$, $(T_1 \mid T_2 \mid T_3 \mid T_4)$, $(T_1 \mid T_1 \mid T_2 \mid T_3)$
Iteration (4): $(T_1 \mid T_2 \mid T_3 \mid T_4)$, $(T_1 \mid T_2 \mid T_3 \mid T_4)$, $(T_1 \mid T_1 \mid T_2 \mid T_3)$, $(T_1 \mid T_1 \mid T_2 \mid T_3)$

Iteration (1): $T_0 \parallel T_1 \parallel T_2 \parallel T_3$
Iteration (2): $(T_0, T_0) \parallel (T_0, T_1) \parallel (T_0, T_2) \parallel (T_0, T_3) \parallel$
$(T_1, T_0) \parallel (T_1, T_1) \parallel (T_1, T_2) \parallel (T_1, T_3) \parallel$
$(T_2, T_0) \parallel (T_2, T_1) \parallel (T_2, T_2) \parallel (T_2, T_3) \parallel$
$(T_3, T_0) \parallel (T_3, T_1) \parallel (T_3, T_2) \parallel (T_3, T_3)$
Iteration (3): $(T_0, T_0, T_0) \parallel (T_1, T_0, T_0) \parallel (T_0, T_1, T_0) \parallel (T_0, T_0, T_1) \parallel$
$(T_1, T_1, T_0) \parallel (T_1, T_0, T_1) \parallel (T_0, T_1, T_1) \parallel (T_1, T_1, T_1)$
Iteration (4): $((T_0, T_0, T_0, T_0) \parallel (T_0, T_0, T_0, T_1) \parallel (T_0, T_0, T_0, T_2) \parallel$
$(T_0, T_0, T_0, T_3) \parallel ....$
(T_3, T_3, T_3, T_3)

With the integration of the decomposed alternative-construct into the transformation scheme description substitution, the following can be said about the first decomposing steps:

1. $(T_0, T_0, ((T_0), T_5[0..1], T_6[0..10])\{C1,C2,C3\}, T_7[1..5])\{C4\} \parallel$
2. $(T_0, T_0, ((T_1), T_5[0..1], T_6[0..10])\{C1,C2,C3\}, T_7[1..5])\{C4\} \parallel$
3. $(T_0, T_0, ((T_2), T_5[0..1], T_6[0..10])\{C1,C2,C3\}, T_7[1..5])\{C4\} \parallel$
4. $(T_0, T_0, ((T_3), T_5[0..1], T_6[0..10])\{C1,C2,C3\}, T_7[1..5])\{C4\} \parallel$
.....
.....

Further unfolding would result in over 37400 transformation sequences and a total appliance of more than 77626 transformations. Estimating that one transformation would take 1 second to be computed, the overall processing time would be 1293 min or 21 hours and 32 min. Following the transformation sequences analysis procedure within the FermaT transformation application test on WSL program specific AST nodes does not reveal any elimination of transformation sequences. This also specifies that the first two transformations within the transformation scheme description are static, its applicability prediction does not change, which can be withdrawn from Table 9.7.

Transformation	AST General Type	AST Specific Type
T_0	T_Assign	
T_1		
T_2	T_Assign	
T_3	T_A_S	
T_4		
T_5	T_Assign, T_Expression, T_Guarded	
T_6	T_Assign, T_Expression, T_Guarded, T_A_S	T_Cond, T_D_If, T_Floop T_Var, T_Where, T_While
T_7	T_Action	T_Proc, T_Funct

TABLE 9.7: Transformation Effects of Case Study 2 Transformation

This stage reveals how thoughtfully the headnode analyser follows this elimination technique. The search depth of applicable transformation sequences plays a major role, because the applicability test of transformations within each transformation sequence could consume quite a long time presenting 37400 transformation sequences. In regard to this, the default search depth of individual transformation sequence is set to two transformations. However this value can be adjusted by utilising the developed tool support of the FCE.

9.7.3 Parallel Transformation Processing Techniques

After the analysis of the transformation scheme description and generated transformation sequences, the evaluation of a suitable parallel transformations processing technique based on the definition of the specified parallel constraints comes into account. Similar to the first case study the overall constraints are parallel transformations processing and linear-line processing, which are presented in the following.

9.7.4 Parallel Transformation Processing Case Study 2

As revealed within the analysis steps, no transformation sequence could be eliminated by not being applicable. Therefore the presented parallel transformations processing technique groups the generated transformation sequences into packages of individual sub-schemes and computes them independently in parallel. Listing 9.12 presents this kind of grouping for case study 2.

```
1 (<T0>,<T0>,(<T1>,(<T1>|<T2>|<T3>|<T4>)[0..3]),<T5>[0..1],<T6>[0..10]){C1,C2,C3},
2         <T7>[1..5]){C4}
3 (<T0>,<T0>,(<T2>,(<T1>|<T2>|<T3>|<T4>)[0..3]),<T5>[0..1],<T6>[0..10]){C1,C2,C3},
4         <T7>[1..5]){C4}
5 (<T0>,<T0>,(<T3>,(<T1>|<T2>|<T3>|<T4>)[0..3]),<T5>[0..1],<T6>[0..10]){C1,C2,C3},
6         <T7>[1..5]){C4}
7 (<T0>,<T0>,(<T4>,(<T1>|<T2>|<T3>|<T4>)[0..3]),<T5>[0..1],<T6>[0..10]){C1,C2,C3},
8         <T7>[1..5]){C4}
```

Listing 9.12: 4 Sub-Schemes of Case Study 2

Further analysis of the similarity of the above sub-schemes reveals that the first two transformations could be applied within the headnode before its submission and assignment to the computing nodes. This technique opens the possibility to detect further parallelism since at this point it is unknown how much parallelisation could be validated by analysing the alternative-construct.

```
1  (<T0>,<T0>,(<T1>,((<T1>|<T2>),(<T1>|<T2>|<T3>|<T4>)[0..2])[0..1]),<T5>[0..1],
2          <T6>[0..10]){C1,C2,C3},<T7>[1..5]){C4}
3  (<T0>,<T0>,(<T1>,((<T3>|<T4>),(<T1>|<T2>|<T3>|<T4>)[0..2])[0..1]),<T5>[0..1],
4          <T6>[0..10]){C1,C2,C3},<T7>[1..5]){C4}
5  (<T0>,<T0>,(<T2>,((<T1>|<T2>),(<T1>|<T2>|<T3>|<T4>)[0..2])[0..1]),<T5>[0..1],
6          <T6>[0..10]){C1,C2,C3},<T7>[1..5]){C4}
7  (<T0>,<T0>,(<T2>,((<T3>|<T4>),(<T1>|<T2>|<T3>|<T4>)[0..2])[0..1]),<T5>[0..1],
8          <T6>[0..10]){C1,C2,C3},<T7>[1..5]){C4}
9  (<T0>,<t0>,(<T3>,((<T1>|<T2>),(<T1>|<T2>|<T3>|<T4>)[0..2])[0..1]),<T5>[0..1],
10         <T6>[0..10]){C1,C2,C3},<T7>[1..5]){C4}
11 (<T0>,<T0>,(<T3>,((<T3>|<T4>),(<T1>|<T2>|<T3>|<T4>)[0..2])[0..1]),<T5>[0..1],
12         <T6>[0..10]){C1,C2,C3},<T7>[1..5]){C4}
13 (<T0>,<T0>,(<T4>,((<T1>|<T2>),(<T1>|<T2>|<T3>|<T4>)[0..2])[0..1]),<T5>[0..1],
14         <T6>[0..10]){C1,C2,C3},<T7>[1..5]){C4}
15 (<T0>,<T0>,(<T4>,((<T3>|<T4>),(<T1>|<T2>|<T3>|<T4>)[0..2])[0..1]),<T5>[0..1],
16         <T6>[0..10]){C1,C2,C3},<T7>[1..5]){C4}
```

Listing 9.13: 8 shortened Sub-Schemes of Case Study 2

Estimating that the parallel processing environment consists of 8 computing nodes, the generated transformation sequences could be further decomposed and grouped to independent transformation sub-schemes, demonstrated in Listing 9.13. However, as each

computing node would still need to compute over 4675 transformation sequences, it seems to be more appropriate to further decompose the transformation scheme description. The alternative-construct specified within this transformation scheme description can be only decomposed into an even number of groups, which in this case would make more sense in concern of even load-balancing of the 8 computing nodes.

9.7.5 Parallel Transformation Task Generation Case Study 2

Similar to case study 1, the next processing step results in the construction of the "*PLACED PAR*" construct which is further utilised to assign the independent transformation tasks to the computing nodes. The development by sharing the TCP/IP communication layer can be followed in Listing 9.14. The next step would eventually result in the acknowledgement of the computing nodes that they can start their parallel transformations processing. As this parallel processing environment only consists of 6 computing nodes, the first 6 tasks are run parallel followed by the last ones.

```
1 PLACED PAR
2   fc-node1 TCP/IP (1, caseStudy2.wsl, task1.tdsl)
3   fc-node2 TCP/IP (2, caseStudy2.wsl, task2.tdsl)
4   fc-node3 TCP/IP (3, caseStudy2.wsl, task3.tdsl)
5   fc-node4 TCP/IP (4, caseStudy2.wsl, task4.tdsl)
6   fc-node5 TCP/IP (5, caseStudy2.wsl, task5.tdsl)
7   fc-node6 TCP/IP (6, caseStudy2.wsl, task6.tdsl)
8   fc-node7 TCP/IP (7, caseStudy2.wsl, task7.tdsl)
9   fc-node8 TCP/IP (8, caseStudy2.wsl, task8.tdsl)
```

Listing 9.14: PLACED PAR Case Study 2

9.7.6 Parallel Processing Results Case Study 2

To conclude, similar to case study 1 and according to the given charts and tables below, by sub-dividing the transformation scheme description into individual sub-schemes the parallel processing results could be reduced by a tremendous amount of time compared to the overall parallel processing time performed by a single computing node. The presented table below illustrates the overall decrease in processing time on over 21 hours on a single computing node. The 1700 % decrease in time has been achieved by grouping and avoiding redundant work. A lot of time saving depends on the adjustment of the applicability check of FermaT transformations and its AST path. The comparison in computing time between a single transformation sequence processing and the grouping technique is presented in Table 9.8 and Figure 9.3.

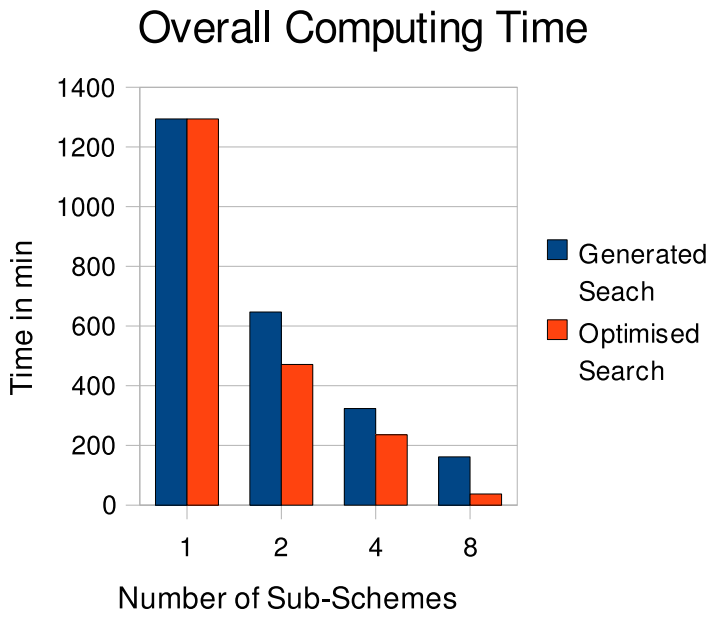


FIGURE 9.3: Overall Computing Time Case Study 2

Overall Processing Time in Minutes				
	Number of Computing Nodes			
	1	**2**	**4**	**8**
Generated Search Space	1293.77	646.887	323.44	161.72
Optimised Search Space	1293.77	471.23	235.63	74.48
Time saved by optimisation	0	175.65	87.81	87.24

TABLE 9.8: Overall Processing Time Case Study 2: 1 - 8 Computing Nodes

9.7.7 Parallel Linear Array Processing Case Study 2

As mentioned the parallel linear-line transformations processing is another concept of parallel transformation processing discussed in detail in Chapter 7. This section illustrates and outlines the presented technique based on case study 2.

The unrolling of transformation scheme descriptions illustrated in the previous sections works in a similar manner to the applied parallel transformation processing. The only difference is that the sequence constructs and alternative-constructs of individual subscheme are assigned to a group of computing nodes, functioning as a linear-line of compute nodes. Although the main purpose of utilising the linear-line transformations processing behaviour results in the batch file processing of more than one WSL program at a time. This process now comes forward, because at the end of each processing-line a lot of WSL program files have to be passed to the next computing node. This process is

specified in Table 9.9 and Listing 9.15 presents the decomposition of the transformation sub-schemes according to the presented laws.

```
1 (<T0>,<T0>,(<T1>,(<T1>|<T2>|<T3>|<T4>)[0..3]),<T5>[0..1],<T6>[0..10]){C1,C2,C3},
2         <T7>[1..5]){C4}
3 (<T0>,<T0>,(<T2>,(<T1>|<T2>|<T3>|<T4>)[0..3]),<T5>[0..1],<T6>[0..10]){C1,C2,C3},
4         <T7>[1..5]){C4}
5 (<T0>,<T0>,(<T3>,(<T1>|<T2>|<T3>|<T4>)[0..3]),<T5>[0..1],<T6>[0..10]){C1,C2,C3},
6         <T7>[1..5]){C4}
7 (<T0>,<T0>,(<T4>,(<T1>|<T2>|<T3>|<T4>)[0..3]),<T5>[0..1],<T6>[0..10]){C1,C2,C3},
8         <T7>[1..5]){C4}
```

Listing 9.15: 4 Transformation Sub-Schemes of Case Study 2

Estimating that the above sub-schemes are parallelised within a linear-line structure the following computing node assignment listed in Table 9.9 ican be utilised.

Sub-Scheme Assignment						
	Computing Node					
	1	2	3	4	5	6
1	(T0)[1..2]	(T1)	(T1|T2|T3|T4)[0..3]	(T5)[0..1]	(T6)[0..10]{C1,C2,C3}	(T7)[1..5]{C4}
2	(T0)[1..2]	(T2)	(T1|T2|T3|T4)[0..3]	(T5)[0..1]	(T6)[0..10]{C1,C2,C3}	(T7)[1..5]{C4}
3	(T0)[1..2]	(T3)	(T1|T2|T3|T4)[0..3]	(T5)[0..1]	(T6)[0..10]{C1,C2,C3}	(T7)[1..5]{C4}
4	(T0)[1..2]	(T4)	(T1|T2|T3|T4)[0..3]	(T5)[0..1]	(T6)[0..10]{C1,C2,C3}	(T7)[1..5]{C4}

Table 9.9: Pipeline Sub-Scheme Assigning Case Study 2

Being aware that the first two transformations of each transformation sub-scheme are the same, they could be grouped into one instance, assigned with a quantifier construct form of "[1..2]" and scheduled on one separate computing node. As stated, a unique by-product of this technique is that all generated sub-schemes have the same length and therefore can be equally assigned to computing nodes. It has to be kept in mind that the more the transformation sub-schemes are decomposed the longer the linear processing is going to be. As long as the number of computing nodes are the same sub-schemes are no further decomposed, the total processing time does not change and therefore it is more a matter of linear-processing line design.

9.7.8 Results and Summary Case Study 2

This case study demonstrated that generating and computing 37400 transformation sequences with over 62600 transformation applications can be a challenging task. The proposed parallel transformations processing environment solves this in a quite reasonable

time. This is mostly due to the fact that not applicable and redundant transformation sequences are eliminated by the presented analysing system. Furthermore the decomposition of transformation scheme descriptions and the presentation of parallelisation techniques solved this enormous task in a reasonable time.

In the search for an appropriate fulfilment of the specified reengineering aim, the transformation sequence presented in Listing 9.16 fulfils this result. Based on its length, it is one of the shortest with 9 transformations. For its successful application the following transformations were evaluated: Twice transformation "*Merge Right*" on the first found WSL AST type "*T_Assign*", followed by the transformation "*Constant Propagation*" and "*Delete All Redundant*" on the AST root node ("//"). The next processing steps would reveal transformation "*Substitute and Delete*" to be an appropriate choice, because it fulfils the specified reengineering constraints, by leaving no WSL specific AST type "*T_Prog*" within the final WSL program source.

```
1
2   < Merge Right @ / 0,1,2,3,6,1,0 / > ,
3   < Merge Right @ / 0,1,2,3,6,1,0 / > ,
4   < Constant Propagation @ // > ,
5   < Delete All Redundant @ // > ,
6   {C1,C2,C3} ,
7   < Substitute and Delete @ /0,1,0/ > ,
8   < Substitute and Delete @ /0,1,0/ > ,
9   < Substitute and Delete @ /0,1,0/ > ,
10  < Substitute and Delete @ /0,1,0/ > ,
11  < Substitute and Delete @ /0,1,0/ > ,
12   {C4}
```

Listing 9.16: Case Study 2 Satisfying Transformation Sequence

The outcome is the WSL program source presented in Listing 9.17 and fulfils **all** specified reengineering constraints.

```
A := ARRAY(10000, 0);
length := 10000;
element := 5001;
init := 1;
FOR i := 1 TO 10000 STEP 1 DO
  A[i] := 10000;
  A[i] := 2 * A[i];
  A[i] := A[i] - i;
  A[i] := A[i] - i;
  A[i] := A[i] + 2 OD;
n := length;
DO swapped := 0;
   FOR i := 1 TO n - 1 STEP 1 DO
     IF A[i + 1] < A[i]
       THEN temp := A[i];
            A[i] := A[i + 1];
            A[i + 1] := temp;
            swapped := 1 FI OD;
   IF swapped = 0 THEN EXIT(1) FI OD;
FOR i := 1 TO length STEP 1 DO
  temp := A[i] + 1; A[i] := temp OD;
low := 1;
high := length;
result := -1;
WHILE result = -1 AND high >= low DO
  mid := high DIV 2;
  mid := -low DIV 2 + mid;
  mid := low + mid;
  IF A[mid] > element
    THEN high := mid - 1
  ELSIF A[mid] < element
    THEN low := mid + 1
    ELSE result := mid FI OD;
PRINT("RESULT: ", result)
```

Listing 9.17: Case Study 2 Final WSL Programe Source

To conclude in the matter of an appropriate parallel transformations processing solution, Figure 9.4 presents a good comparison of both parallel transformations processing techniques described. The chart classifies with Table 9.10 below that the division into individual sub-schemes contributes the most to time saving. The usage of the linear-processing line technique increases the processing time by more than 19 %. This is due to the fact that the process of passing WSL program files between the computing nodes, consumes a lot of processing time.

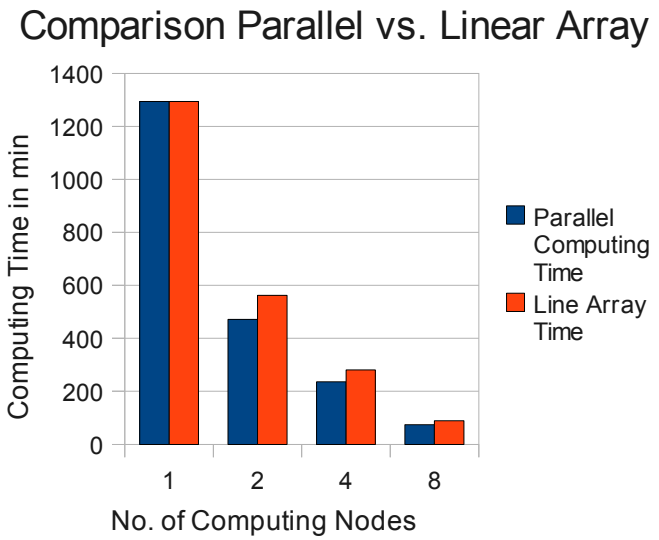


FIGURE 9.4: Comparison of the Overall Computing Time: Case Study 2

Overall Processing Time in Minutes					
		Number of Computing Nodes			
		1	**2**	**4**	**8**
1	Parallel Processing	1293.77	471.23	235.63	74.48
2	Linear Array Computing	1293.77	562.43	281.27	88.97
	Performance: 1 vs. 2		+19.3%	+19.4%	+19.6%

TABLE 9.10: Overall Processing Time Case Study 2: 1 - 8 Computing Nodes

Comparing the overall performance to a single processing environment, Table 9.10 above shows the tremendous computing power which can be achieved by subdividing the transformation scheme descriptions into independent sub-schemes. However analysing the network communication reveals that the standard parallel computing is a much more efficient solution compared to the linear line approach. Due to this fact, the transformation sequence with the number 18586 satisfies all reengineering constraints and is already computed after 58 min on an 8 computing node cluster configuration. In comparison to the linear-line processing it needs more than 71 minutes to be found. On the other hand these real life scenarios also reveal how important it is to analyse any transformation task. The more precisely a transformation scheme is analysed for its transformation applicability check the more efficiently it can be computed. Within

case study 1, the proposed analysing system quickly reveals that many transformation sequences can be removed from the generated reengineering task space. This is not only due to the fact that they are not applicable, it also demonstrated how important it is to have as a fundamental part, an evaluation system which is combined with knowledge, absorbing misleading maintainer decisions. The presented parallel transformations processing theory is in its way unique, because it is fully adjustable either in describing and outlining parallel transformation tasks via the developed formal language or by the employment of the transformation scheme description language and its reengineering constraints. Nonetheless it has demonstrated that with the help of this proposed parallel transformations processing system, any reengineering specific transformation task can be automated, computed and converted in a more efficient solution than on a single system.

9.8 Summary

This chapter presented two case studies, both focused to demonstrate the capability of the proposed parallel transformations framework to parallelise transformation tasks. The first case study, a medium scale example, generates 5460 transformation sequences whereas the second produces over 37400 sequences. Computing each in parallel revealed that with a small amount of parallel processing power, the overall computing time can be tremendously reduced to almost linear speed-up. The proposed parallel transformations framework also revealed how crucial it is, to have as a fundamental part, an analysing and evaluation system. This system, analyses and evaluates transformation tasks, captures information and automatically adds reengineering information to speed-up the parallel computation.

Chapter 10

Conclusion and Future Research

Objectives

- To evaluate and summerise the presented work.
- To discuss the limitations of the presented approach.
- To draw conclusions.
- To propose the future work.

10.1 Summary of the Thesis

This thesis introduces a parallel transformations framework for software migration processes. The system focuses on parallelising transformation tasks for program transformations application. The approach is unique, as it combines todays parallel processing techniques with a program transformation system to a parallel transformations processing environment.

The basis for this achievement is an architectural design of a parallel processing environment based on the Beowulf cluster system, utilising Commercial Off-The-Shelf (COTS) PC components. The result is a system that computes transformation tasks in an adjustable and parallel manner. Unique because an analysing system evaluates specified transformation tasks to produce suitable parallel transformations processing outlines,

based on metrics and constraints. To lead to a successful parallel processing solution, different parallel processing solutions are evaluated and combined within this approach.

To further assist the parallel transformations processing behaviour, a formal language has been specified to describe and outline transformation tasks. Once defined and submitted to the parallel processing environment, the systems headnode analyses and decomposes each transformation process on the basis of parallel transformations laws. Computing nodes independently compute and evaluate assigned tasks. Results are evaluated on computation based reengineering constraints. The features of the presented framework can be summarised as:

- A parallel transformations environment on the basis of a Beowulf cluster architecture.
- An adjustable parallel processing environment in which computing nodes can be dynamically added and removed during system runtime.
- A service oriented parallel processing approach in which system components can be extended by the utilisation of interfaces.
- Development of an analysing system with attributes of:
  - Parallel processing environment analysis.
  - Parallel transformation task analysis.
  - Transformation scheme description analysis.
- Presentation of a formal language to describe and outline parallel transformation tasks.
- Parallel transformation task refinement through constraints.
- Parallel transformation task submission and result evaluation.

The above features have been carefully chosen during the research investigation. The main focus has been laid on the establishment of a parallel transformations processing architecture combination of features: parallel speed-up, flexibility and service oriented design. In regard to these aspects attempts have been made to remove the following limitations from the current FermaT transformation process:

- Sequential to parallel transformations processing through the establishment of the proposed framework.

- Decrease of the enormous generated transformation sequence search-space by the establishment of an analysing system. The system evaluates the generated search-space and eliminates redundant work.
- Speed-up of transformation processes by the presentation of laws to decompose and parallelise transformation scheme descriptions.
- Definition of a parallel transformation task language specification, to automate parallel transformations processes.

10.2 Evaluation

Chapter 1 defined a set of research questions. These questions are used to evaluate the proposed approach.

- *How can automated parallel transformations processing be achieved?*

  The possibility to automatically parallelise transformations processes has been proven to be possible within Chapter 4, Chapter 5 and Chapter 7. It has been achieved by the establishment of a parallel transformations processing framework, specifying an environment which analyses, decomposes and parallelises transformation tasks. Transformation tasks can be outlined by a maintainer with the help of a formal language. Transformation tasks always follow a specific maintenance goal. An analysing system evaluates the presented data and produces parallel transformations processing outlines. The formal language underpins an automated approach as the parallel environment automatically computes specified tasks and forwards the outcome to the user.

- *Which techniques can be utilised to decompose transformation scheme descriptions?*

  The decomposition of transformation scheme descriptions has been achieved by the definition of decomposition laws. Chapter 7 outlines these laws in particular and presents different strategies and examples to decompose transformation scheme descriptions for parallel computation.

- *Can transformation search problems be mapped to a parallel computing environment?*

  The execution of transformation scheme descriptions can result in a tremendous transformation sequence search space. To decrease the overall computation time, Chapter 7 presents a solution to eliminate inapplicable transformation sequence

before the parallel transformations processing starts. To speed-up the computation, the applicable transformation sequence search space is grouped, forwarded and computed in parallel by the computing nodes.

- *How well can parallelisation be integrated within the FermaT transformation system?*

  To present a flexible parallel transformations processing platform, the classical Beowulf architecture seemed to be the most appropriate solution within the current developing stage of the FermaT transformation engine. For this reason, the parallel processing approach adheres to being flexible by designing parallel components and services which can be enhanced during future development stages. Chapter 5 describes these services and outlines their behaviour within the parallel transformations processing domain.

- *How big are the advantages of a parallel program transformations approach against a common one?*

  The advantages of this approach are:

  - The presented parallel transformations processing environment provides a solution to predefine and automatically parallelise and compute transformation task. It allows to completely outline a transformation process before a single transformation is applied. This preserves the user from the whole impact of each particular transformation and its effects.
  - The presented parallel transformations processing environment provides a solution to predefine and automatically parallelise and compute transformation tasks. It allows a transformation process to be completely outlined before a single transformation is applied. The utilisation of transformation scheme descriptions preserves the maintainer from the impact of needing to know each particular transformation and its effects.
  - A combination and utilisation of formal languages allows an auto- mated parallel transformations processing approach to be presented without any user interaction, accept that the maintainer has to submit the transformation task to the specified environment.
  - The presented technique computes and decreases the overall computing time by combining multiple computing resources to a parallel processing platform. By defined interfaces, additional computing resources can be dynamically added to the presented system.
  - To speed-up transformation tasks, an analysing system analyses each task beforehand and eliminates inapplicable transformations. Similar generated

transformation sequences are combined to avoid redundant work and to decrease the processing time.

- By presenting a formal language to describe and outline parallel transformation processes, the maintainer is able to utilise a powerful interface to control and direct parallel transformation processes to computing nodes or utilise specific parallel computing resources.
- Parallel processing and transformation constraints can be utilised to control the behaviour of transformation processes. This allows the analysing system to reject or filter transformation tasks by providing the maintainer with a solution to present applicable or not-applicable processing results.

10.3 Limitation of this Approach

The proposed approach has shown the potential of achieving parallelism within the FermaT transformation system. However there are some limitations within the presented approach.

- The fact that transformation scheme descriptions outline complete transformation processes and that they are based on knowledge of the maintainer means that any incorrect transformation process definition leads to incorrect or slow processing results.
- Similar to the point above, parallel transformation tasks outline the entire parallel process and are based on knowledge of the maintainer. Any incorrect task definition leads to slow processing or incorrect parallel processing results.
- Parallel computation speed mainly depends on utilised hardware components. The faster the utilised PC components can process results, the less the overall computation time will be. Approach and prototype tools have demonstrated the potential of parallelising transformations processes, however the hardware components used within the case studies are not state-of-the-art. Faster processing machines could have possibly increased the specified parallel transformation tasks.
- Due to the fact that the proposed parallel transformations processing architecture is a customisable oriented transformations processing environment, utilised and exchangeable communication systems are not the fastest. However case studies have proven that the communication delay between the computing nodes only plays a minor role.

10.4 Conclusion and Future Work

The presented thesis gave an insight in the applicability of program transformations within the FermaT transformation system. It revealed the difficulties in converting the program transformation process from a sequential into a parallel one. To remedy some of the problems faced, a framework to compute, describe and outline parallel transformation processes is introduced. The development of an analysing system which evaluates suitable parallel processing outlines supports this direction. A prototype tool and evaluation of different parallel transformations processing case studies have shown promising results. This has been the initial step of introducing parallelism to a FermaT transformation engine. However the technique revealed that the success of a transformation task mostly depends on the following points:

- Maintainer knowledge describing and refining parallel transformation tasks and transformation scheme descriptions.
- Constraint definition.
- System previously gained knowledge can enhance the transformation tasks computation and speed-up.

On the basis of the proposed approach the following working packages could further enhance parallel transformations processing:

- Further speed-up of the parallel transformation tasks could be achieved by the implementation of a communication layer directly within the FermaT transformation engine through C. libraries.
- More processing speed could be gained by implementing most of the parallel transformation processing techniques in the programming language C.
- Search tactics to find specified AST types could be improved by the usage of Genetic Algorithms (GA).
- Prediction technique could be improved through the development of techniques which extract knowledge from transformation sequences instead of effects of a single transformation.

Appendix A

FermaT Transformations Descriptions

A.1 Group Delete

Delete All Redundant

The transformation deletes all redundant statements within the selected statements. It consideres a statement as redundant if it calls nothing external and the variables it modifies will all be assigned again before their values are accessed. The transformation can be applied on any type of statement. It belongs to the group "*Delete*" and is available in both versions of the FTE.

```
1  i := 10;
2  j := 20;
3  i := i + j;
4  i := 15
```

Listing A.1: WSL code before the Delete All Redundant transformation

```
1  j := 20;
2  i := 15
```

Listing A.2: WSL code after the Delete All Redundant transformation

Remove All Redundant Variables

The transformation removes all redundant variables within the selected statements. It can be applied on statements which are or which contain the specific type "T_Var". The transformation belongs to the group "*Delete*" and is available in both versions of the FTE.

```
1  VAR <i := 1, j := 2>:
2    WHILE i < 10 DO
3      i := i + j
4    OD
5  ENDVAR;
6  VAR <i := 2, j := 1>:
7    WHILE i < 8 DO
8      i := i + j
9    OD
10 ENDVAR
```

Listing A.3: WSL code before the Remove All Redundant Variables transformation

```
1  VAR <i := 1>:
2    WHILE i < 10 DO
3      i := i + 2
4    OD
5  ENDVAR;
6  VAR <i := 2>:
```

```
7     WHILE i < 8 DO
8       i := i + 1
9     OD
10  ENDVAR
```

Listing A.4: WSL code after the Remove All Redundant Variables transformation

A.2 Group Join

Merge Right

The transformation merges the selected statements into the following statement. It can be applied on statements which have another statement to the right. The transformation belongs to the group "Join" and is available in both versions of the FTE.

```
1  j := 0;
2  k := j;
3  k := k + 1
```

Listing A.5: WSL code before the Merge Right transformation

```
1  j := 0;
2  k := j + 1
```

Listing A.6: WSL code after the Merge Right transformation

A.3 Group Simplify

Constant Propagation

The transformation searches within the selected statements for assignments of constants to variables and propagates the constants through the selected statements by replacing variables with the appropriate value. It can be applied on statements which contain the general type "T_Assign". The transformation belongs to the group "Simplify" and is available in both versions of the FTE.

```
1  i := 5;
2  IF i = 5 THEN
3    j := 0
4  FI
```

Listing A.7: WSL code before the Constant Propagation transformation

```
i := 5;
j := 0
```

Listing A.8: WSL code after the Constant Propagation transformation

Delete Unreachable Code

The transformation deletes unreachable statements within the selected statements. It consideres a statement as unreachable if there is no path in the control flow graph from the start node to the node which contains the selected statement. The transformation can be applied on any type of statement. It belongs to the group "Simplify" and is available in both versions of the FTE.

```
DO
  i := 0;
  EXIT(1);
  i := 1
OD
```

Listing A.9: WSL code before the Delete Unreachable Code transformation

```
DO
  i := 0;
  EXIT(1)
OD
```

Listing A.10: WSL code after the Delete Unreachable Code transformation

Simplify

The transformation simplifies each item within the selected statements as far as possible. It can be applied on any type of statement. The transformation belongs to the group "Simplify" and is available in both versions of the FTE.

```
IF i > 10 THEN
  DO
    j := j + 1
  OD;
  j := 1;
  k := 5
ELSIF i > 20 THEN
  j := 1 + 1;
  k := 5
ELSIF i <= 10 THEN
  j := 1 + 1 + 1;
  k := 5
ELSE
```

```
14    j := 1 + 1 + 1 + 1;
15    k := 5
16  FI
```

Listing A.11: WSL code before the Simplify transformation

```
1   IF i > 10 THEN
2     ABORT;
3     j := 1;
4     k := 5
5   ELSIF i > 20 THEN
6     j := 2;
7     k := 5
8   ELSIF i <= 10 THEN
9     j := 3;
10    k := 5
11  ELSE
12    j := 4;
13    k := 5
14  FI
```

Listing A.12: WSL code after the Simplify transformation

Simplify If

The transformation takes repeated statements out of the selected "IF statement" and simplifies the conditions as far as possible. Additionally, it removes any cases whose conditions imply earlier conditions and "$FALSE$" cases. The transformation can be applied on the specific type "T_Cond". It belongs to the group "Simplify" and is available in both versions of the FTE.

```
1   IF i > 10 THEN
2     j := 1;
3     k := 5
4   ELSIF i > 20 THEN
5     j := 2;
6     k := 5
7   ELSIF i <= 10 THEN
8     j := 3;
9     k := 5
10  ELSE
11    j := 4;
12    k := 5
13  FI
```

Listing A.13: WSL code before the Simplify If transformation

```
1   IF i > 10 THEN
2     j := 1
3   ELSE
4     j := 3
```

```
5  FI;
6  k := 5
```

Listing A.14: WSL code after the Simplify If transformation

Simplify Item

The transformation simplifies the selected item as far as possible. It can be applied on the general types "T_Assign", "T_Expression" and "T_Guarded" and on the specific types "T_A_S", "T_Cond", "T_D_If", "T_Floop", "T_Var", "T_Where" and "T_While". The transformation belongs to the group "Simplify" and is available in both versions of the FTE.

```
1  i := j + j + k + j + k + k
```

Listing A.15: WSL code before the Simplify Item transformation

```
1  i := 3 * (j + k)
```

Listing A.16: WSL code after the Simplify Item transformation

A.4 Group Rewrite

Floop to While

The transformation converts the selected "*DO* loop" into a "*WHILE* loop". It can be applied on the specific type "T_Floop". The transformation belongs to the group "Rewrite" and is available in both versions of the FTE.

```
1  i := 0;
2  DO
3    IF FALSE THEN
4      EXIT(1)
5    FI;
6    i := i + 1
7  OD;
8  j := i
```

Listing A.17: WSL code before the Floop to While transformation

```
1  i := 0;
2  WHILE TRUE DO
3    i := i + 1
4  OD;
5  j := i
```

Listing A.18: WSL code after the Floop to While transformation

Substitute and Delete

The transformation replaces all calls to the selected action, function or procedure with its definition if it is no recursion. Afterwards, it deletes the definition. The transformation can be applied on the general type "T_Action" and on the specific types "T_Funct" and "T_Proc". It belongs to the group "Rewrite" and is available in both versions of the FTE.

```
1  ACTIONS A:
2    A ==
3      i := i + 1;
4      CALL B
5    END
6    B ==
7      j := j + 1;
8      CALL C;
9      CALL D
10   END
11   C ==
12     CALL D
13   END
14   D ==
15     CALL A
16   END
17 ENDACTIONS
```

Listing A.19: WSL code before the Substitute and Delete transformation

```
1  ACTIONS A:
2    A ==
3      i := i + 1;
4      j := j + 1;
5      CALL C;
6      CALL D
7    END
8    C ==
9      CALL D
10   END
11   D ==
12     CALL A
13   END
14 ENDACTIONS
```

Listing A.20: WSL code after the Substitute and Delete transformation

Remove Recursion in Action

The transformation replaces the body of the selected recursive action with a "*DO* loop". It can be applied on the general type "T_Action". The transformation belongs to the group "Rewrite" and is available in both versions of the FTE.

```
ACTIONS PROG:
  PROG ==
    CALL A
  END
  A ==
    IF i = j THEN
      CALL B
    FI;
    i := i + 1;
    CALL A
  END
  B ==
    CALL Z
  END
ENDACTIONS
```

Listing A.21: WSL code before the Remove Recursion in Action transformation

```
ACTIONS PROG:
  PROG ==
    CALL A
  END
  A ==
    DO
      IF i = j THEN
        EXIT(1)
      FI;
      i := i + 1;
      SKIP
    OD;
    CALL B
  END
  B ==
    CALL Z
  END
ENDACTIONS
```

Listing A.22: WSL code after the Remove Recursion in Action transformation

A.5 FermaT Transformation Applicability Check List

FermaT Transformation	**WSL AST Type**
Constant Propagation	T_Assign
Delete All Redundant	Any type of statement
Delete Unreachable Code	T_A_S, T_Statements
Floop to While	T_Floop
Merge Right	T_Assign
Remove All Redundant Variables	T_Statement, T_Statements
Remove Recursion in Action	T_Action
Simplify	T_Assign, T_Expression, T_Guarded
Simplify Item	T_Cond, T_D_If, T_Floop, T_Var, T_Where, T_While,
	T_A_S, T_Assign, T_Assignment, T_Exprssion, T_Condition
Substitute and Delete	T_Action, T_Funct, T_Proc

Table A.1: FermaT Transformation Applicability Check List

Appendix B

WSL AST Types

ID	Name	Syntax Name	General Type	Subtypes
General_Types				
1	T_Statement	Statement	0	
2	T_Expression	Expression	0	
3	T_Condition	Condition	0	
4	T_Definition	Definition	0	
5	T_Lvalue	Lvalue	0	
6	T_Assign	Assign	0	2: 5
7	T_Guarded	Guarded	0	3; 17
8	T_Action	Action	0	9; 17
9	T_Name	Name	0	
Group_Types				
10	T_Expressions	Expressions	0	2
12	T_Lvalues	Lvalues	0	5
13	T_Assigns	Assigns	0	6
14	T_Definitions	Definitions	0	4
15	T_Actions	Actions	0	8
16	T_Guardeds	Guardeds	0	
17	T_Statements	Statements	0	1
Specific_Types				
101	T_A_Proc_Call	A_Proc_Call	1	9; 10; 12
102	T_MW_Proc_Call	MW_Proc_Call	1	9; 10; 12
103	T_X_Proc_Call	X_Proc_Call	1	9; 10
104	T_Stat_Place	Stat_Place	1	
105	T_Stat_Pat_One	Stat_Pat_One	1	
106	T_Stat_Pat_Many	Stat_Pat_Many	1	
107	T_Stat_Pat_Any	Stat_Pat_Any	1	
108	T_Abort	Abort	1	
109	T_Assert	Assert	1	3
110	T_Assignment	Assignment	1	6
111	T_A_S	A_S	1	9; 15
112	T_Call	Call	1	
113	T_Comment	Comment	1	
114	T_Cond	Cond	1	7
115	T_D_If	D_If	1	7
116	T_D_Do	D_Do	1	7
117	T_Exit	Exit	1	
118	T_For	For	1	2; 5; 17
119	T_Foreach_Stat	Foreach_Stat	1	17
120	T_Foreach_Stats	Foreach_Stats	1	17

Table B.1: WSL Syntax

ID	Name	Syntax Name	General Type	Subtypes
121	T_Foreach_TS	Foreach_TS	1	17
122	T_Foreach_TSs	Foreach_TSs	1	17
123	T_Foreach_STS	Foreach_STS	1	17
124	T_Foreach_Expn	Foreach_Expn	1	17
125	T_Foreach_Cond	Foreach_Cond	1	17
126	T_Ateach_Stat	Ateach_Stat	1	17
127	T_Ateach_Stats	Ateach_Stats	1	17
128	T_Ateach_TS	Ateach_TS	1	17
129	T_Ateach_TSs	Ateach_TSs	1	17
130	T_Ateach_STS	Ateach_STS	1	17
131	T_Ateach_Expn	Ateach_Expn	1	17
132	T_Ateach_Cond	Ateach_Cond	1	17
133	T_Floop	Floop	1	17
134	T_Join	Join	1	17
135	T_Pop	Pop	1	5
136	T_Proc_Call	Proc_Call	1	9; 10; 12
137	T_Push	Push	1	2; 5
138	T_Spec	Spec	1	3; 12
139	T_Var	Var	1	13; 17
140	T_Where	Where	1	14; 17
141	T_While	While	1	3; 17
142	T_MW_Proc	MW_Proc	1	9; 12; 17
143	T_MW_Funct	MW_Funct	1	2; 9; 12; 13; 17
144	T_MW_BFunct	MW_BFunct	1	3; 9; 12; 13; 17
145	T_Skip	Skip	1	
146	T_Foreach_NAS	Foreach_NAS	1	17
147	T_Ateach_NAS	Ateach_NAS	1	17
148	T_Foreach_Variable	Foreach_Variable	1	17
149	T_Foreach_Global_Var	Foreach_Global_Var	1	17
150	T_Ateach_Variable	Ateach_Variable	1	17
151	T_Ateach_Global_Var	Ateach_Global_Var	1	17
152	T_Foreach_Lvalue	Foreach_Lvalue	1	17
153	T_Ateach_Lvalue	Ateach_Lvalue	1	17
154	T_For_In	For_In	1	2; 5; 17
155	T_Puthash	Puthash	1	2; 5;
156	T_Print	Print	1	10
157	T_Prinflush	Prinflush	1	10
158	T_Maphash	Maphash	1	2; 9
159	T_Error	Error	1	10
160	T_Stat_Int_One	Stat_Int_One	1	2
161	T_Stat_Int_Any	Stat_Int_Any	1	2
162	T_Stat_Val_One	Stat_Val_One	1	
163	T_Stat_Val_Any	Stat_Val_Any	1	

Table B.2: WSL Syntax

ID	Name	Syntax Name	General Type	Subtypes
166	T_Ifmatch2_Stat	Ifmatch2_Stat	1	1; 17
167	T_Ifmatch2_Expn	Ifmatch2_Expn	1	2; 17
168	T_Ifmatch2_Cond	Ifmatch2_Cond	1	3; 17
169	T_Ifmatch2_Defn	Ifmatch2_Defn	1	4; 17
170	T_Ifmatch2_Lvalue	Ifmatch2_Lvalue	1	5; 17
171	T_Ifmatch2_Assign	Ifmatch2_Assign	1	6; 17
172	T_Ifmatch2_Guarded	Ifmatch2_Guarded	1	7; 17
173	T_Ifmatch2_Action	Ifmatch2_Action	1	8; 17
174	T_Ifmatch2_Stats	Ifmatch2_Stats	1	17
175	T_Ifmatch2_Expns	Ifmatch2_Expns	1	10; 17
177	T_Ifmatch2_Lvalues	Ifmatch2_Lvalues	1	12; 17
178	T_Ifmatch2_Assigns	Ifmatch2_Assigns	1	13; 17
179	T_Ifmatch2_Defns	Ifmatch2_Defns	1	14; 17
180	T_Ifmatch_Stat	Ifmatch_Stat	1	1; 17
181	T_Ifmatch_Expn	Ifmatch_Expn	1	2; 17
182	T_Ifmatch_Cond	Ifmatch_Cond	1	3; 17
183	T_Ifmatch_Defn	Ifmatch_Defn	1	4; 17
184	T_Ifmatch_Lvalue	Ifmatch_Lvalue	1	5; 17
185	T_Ifmatch_Assign	Ifmatch_Assign	1	6; 17
186	T_Ifmatch_Guarded	Ifmatch_Guarded	1	7; 17
187	T_Ifmatch_Action	Ifmatch_Action	1	8; 17
188	T_Ifmatch_Stats	Ifmatch_Stats	1	17
189	T_Ifmatch_Expns	Ifmatch_Expns	1	10; 17
191	T_Ifmatch_Lvalues	Ifmatch_Lvalues	1	12; 17
192	T_Ifmatch_Assigns	Ifmatch_Assigns	1	13; 17
193	T_Ifmatch_Defns	Ifmatch_Defns	1	14; 17
201	T_X_Funct_Call	X_Funct_Call	2	9; 10
202	T_MW_Funct_Call	MW_Funct_Call	2	9; 10
203	T_Expn_Place	Expn_Place	2	
204	T_Var_Place	Var_Place	2	
205	T_Number	Number	2	
206	T_String	String	2	
207	T_Variable	Variable	2	
208	T_Primed_Var	Primed_Var	2	
209	T_Sequence	Sequence	2	10
210	T_Aref	Aref	2	2; 10
211	T_Sub_Seg	Sub_Seg	2	2
212	T_Rel_Seg	Rel_Seg	2	2
213	T_Final_Seg	Final_Seg	2	2
214	T_Funct_Call	Funct_Call	2	9; 10
215	T_Map	Map	2	2; 9
216	T_Reduce	Reduce	2	2; 9

Table B.3: WSL Syntax

ID	Name	Syntax Name	General Type	Subtypes
217	T_Expn_Pat_One	Expn_Pat_One	2	
218	T_Expn_Pat_Many	Expn_Pat_Many	2	
219	T_Expn_Pat_Any	Expn_Pat_Any	2	
220	T_Plus	Plus	2	2
221	T_Minus	Minus	2	2
222	T_Times	Times	2	2
223	T_Divide	Divide	2	2
224	T_Exponent	Exponent	2	2
225	T_Mod	Mod	2	2
226	T_Div	Div	2	2
227	T_If	If	2	2; 3
228	T_Abs	Abs	2	2
229	T_Frac	Frac	2	2
230	T_Int	Int	2	2
231	T_Sgn	Sgn	2	2
232	T_Max	Max	2	2
233	T_Min	Min	2	2
234	T_Intersection	Intersection	2	2
235	T_Union	Union	2	2
236	T_Set_Diff	Set_Diff	2	2
237	T_Powerset	Powerset	2	2
238	T_Set	Set	2	2; 3
239	T_Array	Array	2	2
240	T_Head	Head	2	2
241	T_Tail	Tail	2	2
242	T_Last	Last	2	2
236	T_Set_Diff	Set_Diff	2	2
237	T_Powerset	Powerset	2	2
238	T_Set	Set	2	2; 3
239	T_Array	Array	2	2
240	T_Head	Head	2	2
241	T_Tail	Tail	2	2
242	T_Last	Last	2	2
243	T_Butlast	Butlast	2	2
244	T_Length	Length	2	2
245	T_Reverse	Reverse	2	2
246	T_Concat	Concat	2	2
251	T_Negate	Negate	2	2
252	T_Invert	Invert	2	2
253	T_Struct	Struct	2	2; 9
254	T_Get_n	Get_n	2	2
255	T_Get	Get	2	2
256	T_Gethash	Gethash	2	2

Table B.4: WSL Syntax

ID	Name	Syntax Name	General Type	Subtypes
257	T_Hash_Table	Hash_Table	2	
258	T_Slength	Slength	2	2
259	T_Substr	Substr	2	10
260	T_Index	Index	2	10
261	T_Expn_Int_One	Expn_Int_One	2	2
262	T_Expn_Int_Any	Expn_Int_Any	2	2
263	T_Expn_Val_One	Expn_Val_One	2	
264	T_Expn_Val_Any	Expn_Val_Any	2	
265	T_Fill2_Stat	Fill2_Stat	2	1
266	T_Fill2_Expn	Fill2_Expn	2	2
267	T_Fill2_Cond	Fill2_Cond	2	3
268	T_Fill2_Defn	Fill2_Defn	2	4
269	T_Fill2_Lvalue	Fill2_Lvalue	2	5
270	T_Fill2_Assign	Fill2_Assign	2	6
271	T_Fill2_Guarded	Fill2_Guarded	2	7
272	T_Fill2_Action	Fill2_Action	2	8
273	T_Fill2_Stats	Fill2_Stats	2	17
274	T_Fill2_Expns	Fill2_Expns	2	10
276	T_Fill2_Lvalues	Fill2_Lvalues	2	12
277	T_Fill2_Assigns	Fill2_Assigns	2	13
278	T_Fill2_Defns	Fill2_Defns	2	14
281	T_Fill_Stat	Fill_Stat	2	1
282	T_Fill_Expn	Fill_Expn	2	2
283	T_Fill_Cond	Fill_Cond	2	3
284	T_Fill_Defn	Fill_Defn	2	4
285	T_Fill_Lvalue	Fill_Lvalue	2	5
286	T_Fill_Assign	Fill_Assign	2	6
287	T_Fill_Guarded	Fill_Guarded	2	7
288	T_Fill_Action	Fill_Action	2	8
289	T_Fill_Stats	Fill_Stats	2	17
290	T_Fill_Expns	Fill_Expns	2	10
292	T_Fill_Lvalues	Fill_Lvalues	2	12
293	T_Fill_Assigns	Fill_Assigns	2	13
294	T_Fill_Defns	Fill_Defns	2	14
301	T_X_BFunct_Call	X_BFunct_Call	3	9; 10
302	T_MW_BFunct_Call	MW_BFunct_Call	3	9; 10
303	T_Cond_Place	Cond_Place	3	
304	T_BFunct_Call	BFunct_Call	3	9; 10
305	T_Cond_Pat_One	Cond_Pat_One	3	
306	T_Cond_Pat_Many	Cond_Pat_Many	3	
307	T_Cond_Pat_Any	Cond_Pat_Any	3	
308	T_True	True	3	
309	T_False	False	3	

Table B.5: WSL Syntax

ID	Name	Syntax Name	General Type	Subtypes
310	T_And	And	3	3
311	T_Or	Or	3	3
312	T_Not	Not	3	3
313	T_Equal	Equal	3	2
314	T_Less	Less	3	2
315	T_Greater	Greater	3	2
316	T_Less_Eq	Less_Eq	3	2
317	T_Greater_Eq	Greater_Eq	3	2
318	T_Not_Equal	Not_Equal	3	2
319	T_Even	Even	3	2
320	T_Odd	Odd	3	2
321	T_Empty	Empty	3	2
322	T_Subset	Subset	3	2
323	T_Member	Member	3	2
324	T_Forall	Forall	3	3; 12
325	T_Exists	Exists	3	3; 12
326	T_Implies	Implies	3	3
327	T_Sequenceq	Sequenceq	3	2
328	T_Numberq	Numberq	3	2
329	T_Stringq	Stringq	3	2
330	T_In	In	3	2
331	T_Not_In	Not_In	3	2
332	T_Cond_Int_One	Cond_Int_One	3	2
333	T_Cond_Int_Any	Cond_Int_Any	3	2
334	T_Cond_Val_One	Cond_Val_One	3	
335	T_Cond_Val_Any	Cond_Val_Any	3	
401	T_Proc	Proc	4	9; 12; 17
402	T_Funct	Funct	4	2; 9; 12; 13
403	T_BFunct	BFunct	4	3; 9; 12; 13
404	T_Defn_Pat_One	Defn_Pat_One	4	
405	T_Defn_Pat_Many	Defn_Pat_Many	4	
406	T_Defn_Pat_Any	Defn_Pat_Any	4	
407	T_Defn_Int_One	Defn_Int_One	4	2
408	T_Defn_Int_Any	Defn_Int_Any	4	2
409	T_Defn_Val_One	Defn_Val_One	4	
410	T_Defn_Val_Any	Defn_Val_Any	4	
501	T_Var_Lvalue	Var_Lvalue	5	
502	T_Aref_Lvalue	Aref_Lvalue	5	5; 10
503	T_Sub_Seg_Lvalue	Sub_Seg_Lvalue	5	2; 5
504	T_Rel_Seg_Lvalue	Rel_Seg_Lvalue	5	2; 5
505	T_Final_Seg_Lvalue	Final_Seg_Lvalue	5	2; 5
506	T_Lvalue_Pat_One	Lvalue_Pat_One	5	

Table B.6: WSL Syntax

ID	Name	Syntax Name	General Type	Subtypes
507	T_Lvalue_Pat_Many	Lvalue_Pat_Many	5	
508	T_Lvalue_Pat_Any	Lvalue_Pat_Any	5	
509	T_Struct_Lvalue	Struct_Lvalue	5	5; 9
510	T_Lvalue_Int_One	Lvalue_Int_One	5	2
511	T_Lvalue_Int_Any	Lvalue_Int_Any	5	2
512	T_Lvalue_Val_One	Lvalue_Val_One	5	
513	T_Lvalue_Val_Any	Lvalue_Val_Any	5	
601	T_Assign_Pat_One	Assign_Pat_One	6	
602	T_Assign_Pat_Any	Assign_Pat_Any	6	
603	T_Assign_Pat_Many	Assign_Pat_Many	6	
604	T_Assign_Int_One	Assign_Int_One	6	2
605	T_Assign_Int_Any	Assign_Int_Any	6	2
606	T_Assign_Val_One	Assign_Val_One	6	
607	T_Assign_Val_Any	Assign_Val_Any	6	
701	T_Guarded_Pat_One	Guarded_Pat_One	7	
702	T_Guarded_Pat_Any	Guarded_Pat_Any	7	
703	T_Guarded_Pat_Many	Guarded_Pat_Many	7	
704	T_Guarded_Int_One	Guarded_Int_One	7	2
705	T_Guarded_Int_Any	Guarded_Int_Any	7	2
706	T_Guarded_Val_One	Guarded_Val_One	7	
707	T_Guarded_Val_Any	Guarded_Val_Any	7	
801	T_Action_Pat_One	Action_Pat_One	8	
802	T_Action_Pat_Any	Action_Pat_Any	8	
803	T_Action_Pat_Many	Action_Pat_Many	8	
804	T_Action_Int_One	Action_Int_One	8	2
805	T_Action_Int_Any	Action_Int_Any	8	2
806	T_Action_Val_One	Action_Val_One	8	
807	T_Action_Val_Any	Action_Val_Any	8	
901	T_Name_Pat_One	Name_Pat_One	9	
902	T_Name_Int_One	Name_Int_One	9	2
903	T_Name_Val_One	Name_Val_One	9	

Table B.7: WSL Syntax

Appendix C

FermaT Cluster Environment (FCE) UML Diagrams

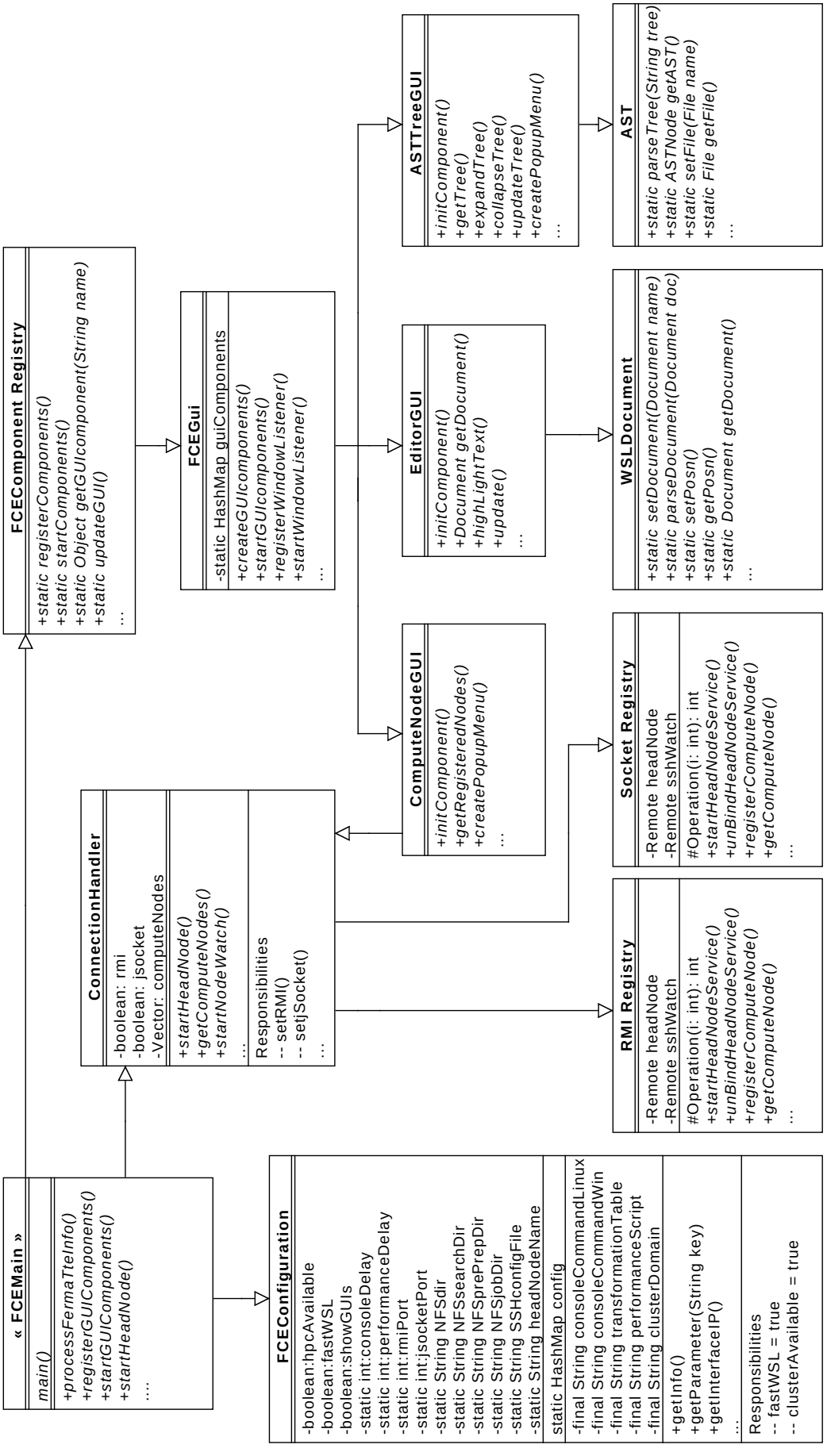


FIGURE C.1: FCE Main Classes Diagram

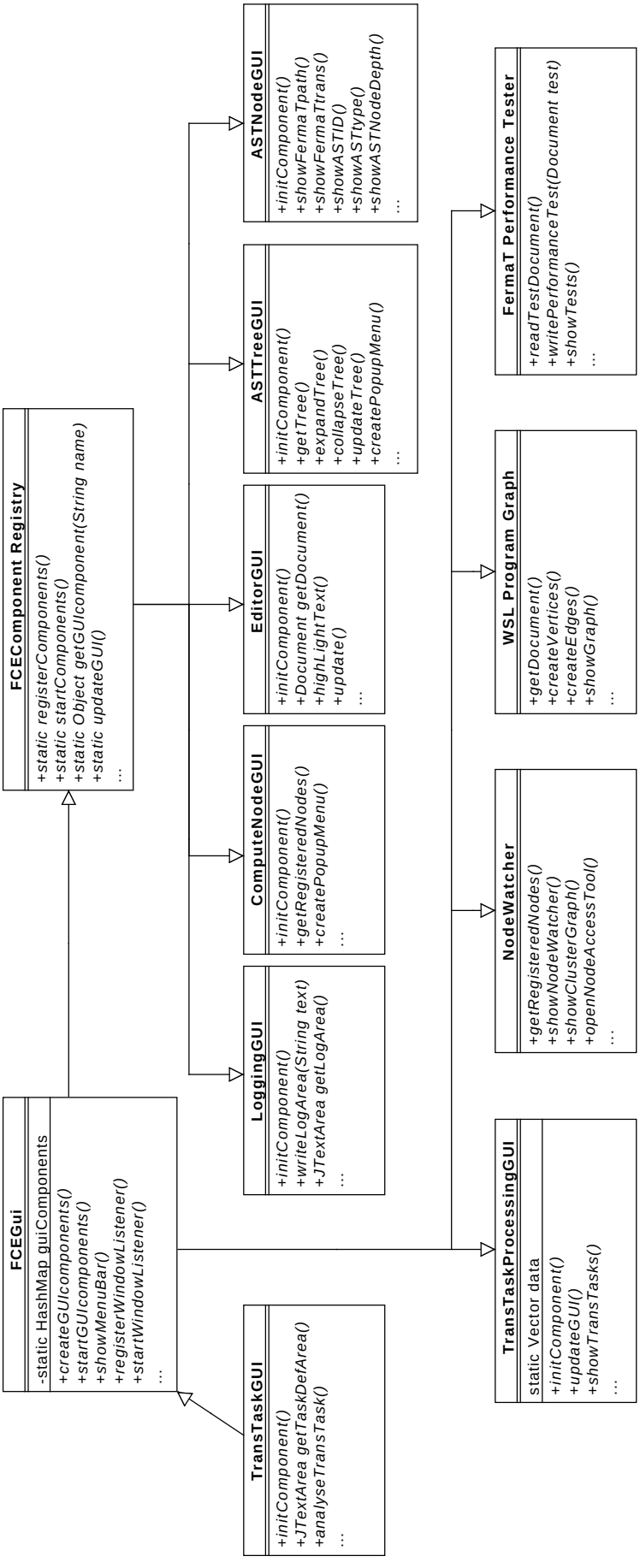


FIGURE C.2: FCE GUIs Class Diagram

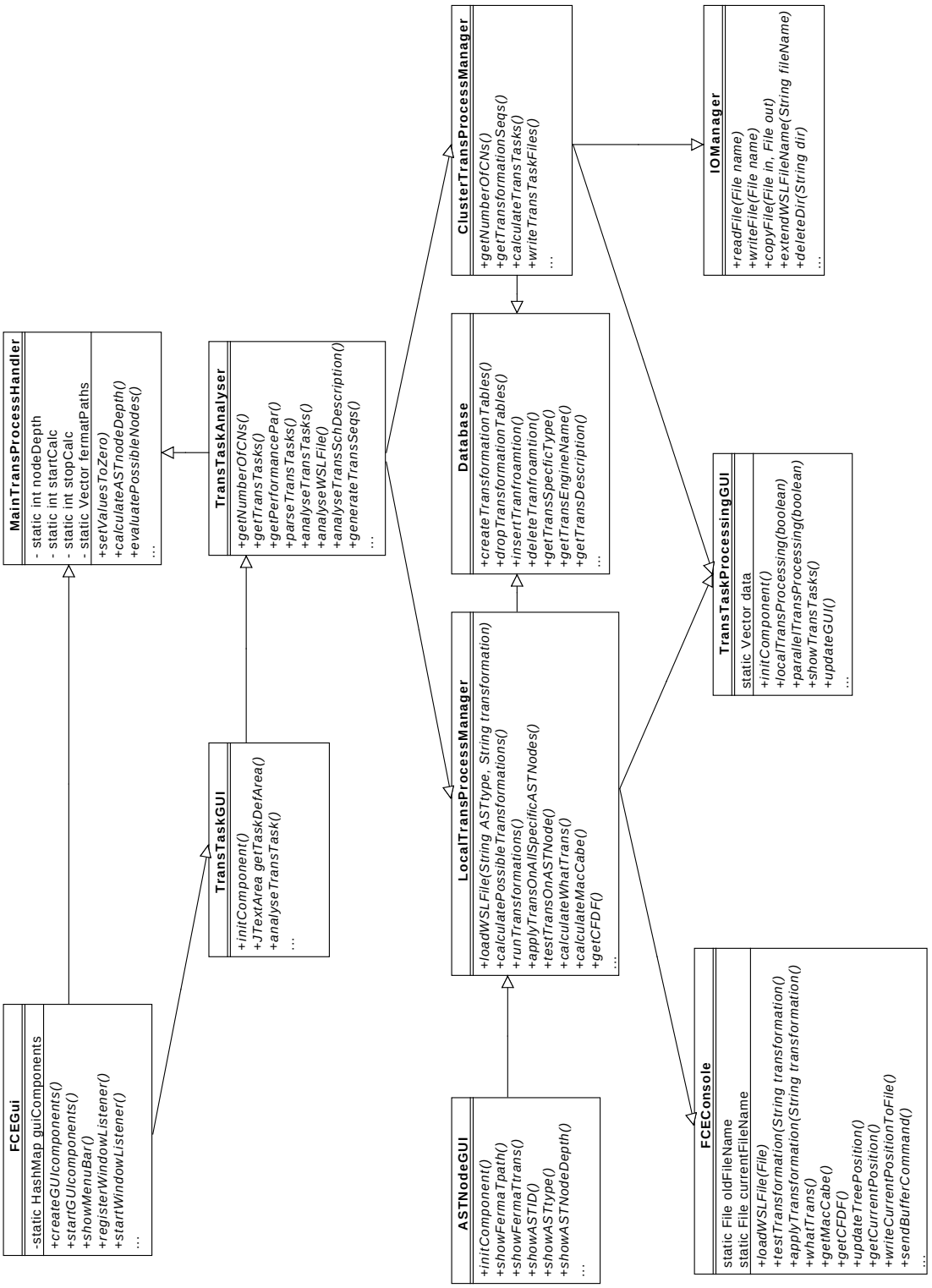


FIGURE C.3: FCE Transformation Processing Class Diagram

Appendix D

Case Study 1 WSL Code

D.1 Case Study 1: Initial WSL Program P_0

```
ACTIONS PROG:
  PROG ==
    i := 55;
    j := k;
    IF k < 0 THEN
      j := j * 5;
      CALL C
    ELSIF k < 25 THEN
      j := j * 2;
      CALL C
    FI;
    CALL A
  END
  A ==
    IF i > j THEN
      k := k + i;
      CALL C
    FI;
    CALL B
  END
  B ==
    IF j < 75 THEN
      j := j + 5;
      k := k * 2;
      CALL B
    FI;
    CALL C
  END
  C ==
    i := 0;
    CALL Z
  END
ENDACTIONS
```

Listing D.1: Case Study 1: Initial WSL Program P_0

D.2 Case Study 1: Final WSL Program P_n

```
1  i := 55;
2  j := k;
3  IF k < 0
4    THEN j := j * 5; i := 0
5  ELSIF k < 25
6    THEN j := j * 2; i := 0
7    ELSE IF i > j
8           THEN k := k + i; i := 0
9           ELSE DO IF j < 75 THEN j := j + 5; k := k * 2 ELSE EXIT(1) FI
10               OD;
11               i := 0 FI FI
```

Listing D.2: Case Study 1: Final WSL Program P_n

Appendix E

Case Study 2 WSL Code

E.1 Case Study 2: Initial WSL Program P_0

```
BEGIN
  A := ARRAY(10000, 0);
  length := 0;
  element := 0;
  length := 10000;
  element := 5001;
  init := 1;
  FOR i := 1 TO length STEP 1 DO
    IF init = 1 THEN
      IF FALSE THEN
        length := 0;
        element := 0
      FI;
      A[i] := length;
      A[i] := A[i] * 2;
      A[i] := A[i] - i;
      A[i] := A[i] - i;
      A[i] := A[i] + 2;
    ELSIF init = 0 THEN
      A[i] := length DIV 2;
      A[i] := A[i] - 1
    FI
  OD;
  SORT( VAR );
  INCREASEALL( VAR );
  SEARCH( VAR );
  PRINT("RESULT: ", result)
WHERE
  PROC SORT( VAR ) ==
    n := 0;
    swapped := 0;
    n := length;
    DO
      swapped := 0;
      FOR i := 1 TO n - 1 STEP 1 DO
        n := length;
        IF A[i] > A[i + 1] THEN
          SWAP( VAR );
          swapped := 1
        FI
      OD;
      IF swapped = 0 THEN
        EXIT(1)
      FI
    OD
  END
  PROC INCREASEALL( VAR ) ==
    FOR i := 1 TO length STEP 1 DO
      INCREASE( VAR )
    OD
  END
  PROC SEARCH( VAR ) ==
    low := 0;
```

```
54      high := 0;
55      result := 0;
56      low := 1;
57      high := length;
58      result := -1;
59      WHILE low <= high AND result = -1 DO
60        mid := high DIV 2;
61        mid := mid - low DIV 2;
62        mid := mid + low;
63        IF A[mid] > element THEN
64          high := mid - 1
65        ELSIF A[mid] < element THEN
66          low := mid + 1
67        ELSIF 5 > 7 THEN
68          result := mid
69        ELSIF FALSE THEN
70          ABORT
71        ELSE
72          result := mid
73        FI
74      OD
75    END
76    PROC SWAP( VAR ) ==
77      temp := 0;
78      temp := A[i];
79      A[i] := A[i + 1];
80      A[i + 1] := temp
81    END
82    PROC INCREASE( VAR ) ==
83      temp := 0;
84      temp := A[i] + 1;
85      A[i] := temp
86    END
87  END
```

Listing E.1: Case Study 2: Initial WSL Program P_0

E.2 Case Study 2: Final WSL Program P_n

```
A := ARRAY(10000, 0);
length := 10000;
element := 5001;
init := 1;
FOR i := 1 TO 10000 STEP 1 DO
  A[i] := 10000;
  A[i] := 2 * A[i];
  A[i] := A[i] - i;
  A[i] := A[i] - i;
  A[i] := A[i] + 2 OD;
n := length;
DO swapped := 0;
   FOR i := 1 TO n - 1 STEP 1 DO
     IF A[i + 1] < A[i]
       THEN temp := A[i];
            A[i] := A[i + 1];
            A[i + 1] := temp;
            swapped := 1 FI OD;
   IF swapped = 0 THEN EXIT(1) FI OD;
FOR i := 1 TO length STEP 1 DO
  temp := A[i] + 1; A[i] := temp OD;
low := 1;
high := length;
result := -1;
WHILE result = -1 AND high >= low DO
  mid := high DIV 2;
  mid := -low DIV 2 + mid;
  mid := low + mid;
  IF A[mid] > element
    THEN high := mid - 1
  ELSIF A[mid] < element
    THEN low := mid + 1
    ELSE result := mid FI OD;
PRINT("RESULT: ", result)
```

Listing E.2: Case Study 2: Final WSL Program P_n

Appendix F

FermaT Transformations Performance Test

F.1 FermaT Transformations Performance XML Specification

```
<?xml version="1.0"?>
<FermaT_Transformations>
        <Group>
                <Name>Group Delete</Name>
                <Transformation>
                        <Name>Delete All Redundant</Name>
                        <FermaT_Engine_Name>//T/R_/Delete_/All_/Redundant
                        </FermaT_Engine_Name>
                        <WSL_Test_File>delete_all_redundant_example_1.wsl
                        </WSL_Test_File>
                        <AST_Path>null</AST_Path>
                </Transformation>
                <Transformation>
                        <Name>Remove All Redundant Variables</Name>
                        <FermaT_Engine_Name>//T/R_/Remove_/All_/Redundant_/Vars
                        </FermaT_Engine_Name>
                        <WSL_Test_File>
   remove_all_redundant_variables_example_1.wsl
                        </WSL_Test_File>
                        <AST_Path>null</AST_Path>
                </Transformation>
        </Group>
        <Group>
                <Name>Join</Name>
                <Transformation>
                        <Name>Merge Right</Name>
                        <FermaT_Engine_Name>//T/R_/Merge_/Right
                        </FermaT_Engine_Name>
                        <WSL_Test_File>merge_right_example_1.wsl
                        </WSL_Test_File>
                        <AST_Path>@DOWN @DOWN</AST_Path>
                </Transformation>
        </Group>
        <Group>
                <Name>Rewrite</Name>
                <Transformation>
                        <Name>Floop to While</Name>
                        <FermaT_Engine_Name>//T/R_/Floop_/To_/While
                        </FermaT_Engine_Name>
                        <WSL_Test_File>merge_right_example_1.wsl
                        </WSL_Test_File>
                        <AST_Path>@DOWN @RIGHT</AST_Path>
                </Transformation>
                <Transformation>
                        <Name>Substitute and Delete</Name>
                        <FermaT_Engine_Name>//T/R_/Substitute_/And_/Delete_/List
                        </FermaT_Engine_Name>
                        <WSL_Test_File>substitute_and_delete_example_1.wsl
                        </WSL_Test_File>
                        <AST_Path>@DOWN @DOWN @RIGHT @DOWN</AST_Path>
                </Transformation>
```

```
51                  <Transformation>
52                          <Name>Remove Recursion in Action</Name>
53                          <FermaT_Engine_Name>//T/R_/Recursion_/To_/Loop
54                          </FermaT_Engine_Name>
55                          <WSL_Test_File>remove_recursion_in_action_example_2.wsl
56                          </WSL_Test_File>
57                          <AST_Path>@DOWN @DOWN @RIGHT @DOWN @RIGHT</AST_Path>
58                  </Transformation>
59          </Group>
60          <Group>
61                  <Name>Simplify</Name>
62                  <Transformation>
63                          <Name>Constant Propagation</Name>
64                          <FermaT_Engine_Name>//T/R_/Constant_/Propagation
65                          </FermaT_Engine_Name>
66                          <WSL_Test_File>constant_propagation_example_1.wsl
67                          </WSL_Test_File>
68                          <AST_Path>@DOWN @DOWN</AST_Path>
69                  </Transformation>
70                  <Transformation>
71                          <Name>Delete Unreachable Code</Name>
72                          <FermaT_Engine_Name>//T/R_/Delete_/Unreachable_/Code
73                          </FermaT_Engine_Name>
74                          <WSL_Test_File>delete_unreachable_code_example_1.wsl
75                          </WSL_Test_File>
76                          <AST_Path>null</AST_Path>
77                  </Transformation>
78                  <Transformation>
79                          <Name>Simplify</Name>
80                          <FermaT_Engine_Name>//T/R_/Simplify
81                          </FermaT_Engine_Name>
82                          <WSL_Test_File>simplify_example_2-0.wsl
83                          </WSL_Test_File>
84                          <AST_Path>null</AST_Path>
85                  </Transformation>
86                  <Transformation>
87                          <Name>Simplify If</Name>
88                          <FermaT_Engine_Name>//T/R_/Simplify_/If
89                          </FermaT_Engine_Name>
90                          <WSL_Test_File>simplify_if_example_2-0.wsl
91                          </WSL_Test_File>
92                          <AST_Path>@DOWN</AST_Path>
93                  </Transformation>
94                  <Transformation>
95                          <Name>Simplify Item</Name>
96                          <FermaT_Engine_Name>//T/R_/Simplify_/Item
97                          </FermaT_Engine_Name>
98                          <WSL_Test_File>simplify_item_example_1.wsl
99                          </WSL_Test_File>
100                         <AST_Path>@DOWN @DOWN @DOWN @RIGHT</AST_Path>
101                 </Transformation>
102         </Group>
103 </FermaT_Transformations>
```

Listing F.1: FermaT Transformations Performance XML Specification

F.2 Computing Node Performance Test Example

```
FermaT_Trans_Performance_Test>
     <Node_IP>192.168.1.76</Node_IP>
     <Transformation>
          <FermaT_Engine_Name>//T/R_/Delete_/All_/Redundant</FermaT_Engine_Name>
          <Processing_Time>10</Processing_Time>
     </Transformation>
     <Transformation>
          <FermaT_Engine_Name>//T/R_/Remove_/All_/Redundant_/Vars</
    FermaT_Engine_Name>
          <Processing_Time>11</Processing_Time>
     </Transformation>
     <Transformation>
          <FermaT_Engine_Name>//T/R_/Merge_/Right</FermaT_Engine_Name>
          <Processing_Time>17</Processing_Time>
     </Transformation>
     <Transformation>
          <FermaT_Engine_Name>//T/R_/Floop_/To_/While</FermaT_Engine_Name>
          <Processing_Time>10</Processing_Time>
     </Transformation>
     <Transformation>
          <FermaT_Engine_Name>//T/R_/Substitute_/And_/Delete_/List</
    FermaT_Engine_Name>
          <Processing_Time>10</Processing_Time>
     </Transformation>
     <Transformation>
          <FermaT_Engine_Name>//T/R_/Recursion_/To_/Loop</FermaT_Engine_Name>
          <Processing_Time>10</Processing_Time>
     </Transformation>
     <Transformation>
          <FermaT_Engine_Name>//T/R_/Constant_/Propagation</FermaT_Engine_Name>
          <Processing_Time>10</Processing_Time>
     </Transformation>
     <Transformation>
          <FermaT_Engine_Name>//T/R_/Delete_/Unreachable_/Code</
    FermaT_Engine_Name>
          <Processing_Time>10</Processing_Time>
     </Transformation>
     <Transformation>
          <FermaT_Engine_Name>//T/R_/Simplify</FermaT_Engine_Name>
          <Processing_Time>20</Processing_Time>
     </Transformation>
     <Transformation>
          <FermaT_Engine_Name>//T/R_/Simplify_/If</FermaT_Engine_Name>
          <Processing_Time>10</Processing_Time>
     </Transformation>
     <Overall_Time>
          <Processing_Time>118</Processing_Time>
     </Overall_Time>
</FermaT_Trans_Performance_Test>
```

Listing F.2: Computing Node Performance Test Example

References

[1] M. P. Ward, "Pigs from sausages? reengineering from assembler to C via fermaT transformations," *Sci. Comput. Program*, vol. 52, pp. 213–255, 2004.

[2] "Assembler to C migration using the fermaT transformation system," Aug. 30 1999.

[3] M. Baker, R. Buyya, and D. C. Hyde, "Cluster computing: A high-performance contender," *CoRR*, vol. cs.DC/0009020, 2000. informal publication.

[4] M. Warren, D. J. Becker, M. P. Goda, J. K. Salmon, and T. Sterling, "Parallel supercomputing with commodity components," in *In International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA*, pp. 1372–1381, 1997.

[5] M. P. Ward, H. Zedan, and T. Hardcastle, "Legacy assembler reengineering and migration," in *ICSM*, pp. 157–166, IEEE Computer Society, 2004.

[6] S. Natelbergr, *Constraint Based Program Transformation Theory.* PhD thesis, De Montfort University, 2009.

[7] S. Bendifallah and W. Scacchi, "Understanding software maintenance work," *IEEE Transactions on Software Engineering*, vol. 13, pp. 311–323, 1987.

[8] I. C. S. A. J. T. F. on Computing Curricula, "Software engineering: Final report," tech. rep., 2004.

[9] F. P. Brooks, Jr., *The Mythical Man Month: Essays on Software Engineering.* Addison-Wesley, first ed., 1975.

[10] J. N. Buxton and B. Randell, "Software engineering techniques report of a conference sponsored by the NATO science committee rome italy 27th-31st october 1969."

[11] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[12] H. M. Sneed, "Economics of software re-engineering," *Journal of Software Maintenance: Research and Practice*, vol. 3(3), pp. 163–182, September 1991.

[13] H. Sneed, "Planning the reengineering of legacy systems," *IEEE Software*, vol. 12, pp. 24–33, Jan. 1995.

[14] S. Rugaber, "Program comprehension for reverse engineering," in *AAAI Workshop on AI and Automated Program Understanding*, 1992.

[15] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, (New York, NY, USA), pp. 73–87, ACM, 2000.

[16] B. W. Boehm, "Seven basic principles of software engineering," *The Journal of Systems and Software*, vol. 3, pp. 3–24, Mar. 1983.

[17] E. Visser, "Stratego: A language for program transformation based on rewriting strategies - system description of stratego 0.5," in *Rewriting Techniques and Applications (RTA01), volume 2051 of Lecture Notes in Computer Science*, pp. 357–361, Springer-Verlag, 2001.

[18] M. Ward and K. Bennett, "A practical program transformation system for reverse engineering," in *WCRE*, pp. 212–221, 1993.

[19] E. Visser, Z. el Abidine Benaissa, and A. Tolmach, "Building program optimizers with rewriting strategies," *ACM SIGPLAN Notices*, vol. 34, pp. 13–26, Jan. 1999.

[20] B. Gifford and W. Harrison, "pRETS: A parallel reverse-engineering tool set for the adaption of sequential programs," in *Proceedings of the International Conference on Software Maintenance 1990*, pp. 344–346, IEEE, IEEE Computer Society Press, 1990.

[21] I. D. Baxter, C. Pidgeon, and M. Mehlich, "Dms®: Program transformations for practical scalable software evolution," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, (Washington, DC, USA), pp. 625–634, IEEE Computer Society, 2004.

[22] P. H. Andersen, J. Pizzi, R. Zhu, Y. Cao, D. J. Bagert, J. K. Antonio, F. Lott, and J. C. Grieger, "Evaluation of a methodology for the reverse engineering and parallelization of sequential code," in *PDSE*, pp. 124–133, 1999.

[23] H. M. Sneed, "Encapsulating legacy software for use in client/server systems," in *WCRE*, p. 104, 1996.

[24] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS®: Program transformations for practical scalable software evolution," in *ICSE*, pp. 625–634, IEEE Computer Society, 2004.

[25] D. J. McConnell, B. Lewis, and L. Gray, "Reengineering a single threaded embedded missile application onto a parallel processing platform using metaH," *Real-Time Systems*, vol. 14, no. 1, pp. 7–20, 1998.

[26] M. Englehart and M. Jackson, "ControlH: A specification language and code generator for real-time GN&C applications," tech. rep., Honeywell Technology Center, 1995.

[27] "The fermat assembler re-engineering workbench," in *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, (Washington, DC, USA), p. 659, IEEE Computer Society, 2001.

[28] D. Fatiregun, M. Harman, and R. M. Hierons, "Evolving transformation sequences using genetic algorithms," in *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, (Washington, DC, USA), pp. 66–75, IEEE Computer Society, 2004.

[29] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[30] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *In Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 1–9, ACM Press, 1999.

[31] S. Li, *A Program Transformation Step Prediction Based Re-engineering Approach*. PhD thesis, Software Technology Research Laboratory, De Montfort University,, 2007.

[32] J. Dongarra, A. Geist, R. Manchek, and W. Jiang, "Using pvm 3.0 to run grand challenge applications on a heterogeneous network of parallel computers," in *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pp. 873–877, SIAM Publications, 1992.

[33] R. Buyya, *High Performance Cluster Computing: Architectures and Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999.

[34] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, 2 ed., January 2003.

[35] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[36] C. U. C. for Advanced Computing, "Flynn taxonomy."

[37] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, pp. 33–38, 2008.

[38] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *SJCC*, 1967.

[39] G. Argentini, "A generalization of amdahl's law and relative conditions of parallelism," *CoRR*, vol. cs.DC/0209029, 2002.

[40] Y. Shi, "Reevaluating amdahl's law and gustafson's law," October 1996.

[41] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, 1988.

[42] P. Umnutkittikul, "Investigation of dynamic process planning using computer simulation "arena"." 2008.

[43] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[44] G. Bell and J. Gray, "High performance computing: Crays, clusters, and centers. what next?," Aug. 11 2001.

[45] L.-H. project, "Linux high-availability cluster manager." www, January 2010.

[46] Oracle, "Oracle cluster manager for solaris." www, 2010.

[47] Symantec, "Veritas global cluster manager." www, September 9 2010.

[48] M. P. Ward, "Language oriented programming," *SoftwareConcepts and Tools*, vol. 15, pp. 147–161, 1995.

[49] M. Ward and M. P. Ward, "Assembler to c migration using the fermat transformation system," in *In IEEE International Conference on Software Maintenance (ICSM99*, pp. 67–76, IEEE Computer Society Press, 1999.

[50] M. Ward and H. Zedan, "Analysing and abstracting legacy assembler code via conditioned semantic slicing.," tech. rep., Software Technology Research Laboratory, De Montfort University, September 2006.

[51] M. P. Ward and H. Zedan, "MetaWSL and meta-transformations in the fermaT transformation system," in *COMPSAC*, pp. 233–238, IEEE Computer Society, 2005.

[52] M. P. Ward, "Language oriented programming," *SoftwareConcepts and Tools*, vol. 15, pp. 147–161, 1995.

[53] C. R. Karp, *Languages with Expressions of Infinite Length.* Amsterdam: North-Holland, 1964.

[54] C. Morgan, *Programming from Specifications.* Prentice–Hall, 1990.

[55] M. P. Ward and H. Zedan, "Slicing as a program transformation," *ACM Trans. Program. Lang. Syst*, vol. 29, no. 2, 2007.

[56] E. W. Dijkstra, "The humble programmer," *Communications of the ACM*, vol. 15, pp. 859–866, Oct. 1972.

[57] E. W. Dijkstra, *A Discipline of Programming.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1976.

[58] M. Lehman, D. Perry, J. Ramil, W. Turski, and P. Wernick, "Metrics and laws of software evolution–the nineties view," in *Proceedings IEEE International Software Metrics Symposium (METRICS'97)*, (Los Alamitos CA), pp. 20–32, IEEE Computer Society Press, 1997.

[59] J. Wileden, "Programming languages and software engineering: past, present and future," *ACM Comput. Surv.*, p. 202.

[60] K. Cremer, A. Marburger, and B. Westfechtel, "Graph-based tools for re-engineering," *Journal of Software Maintenance*, vol. 14, no. 4, pp. 257–292, 2002.

[61] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "Beowulf: A parallel workstation for scientific computation," in *In Proceedings of the 24th International Conference on Parallel Processing*, pp. 11–14, CRC Press, 1995.

[62] D. Daniel and C. Hyde, "Introduction to the programming language occam," 1995.

[63] MPI Forum, "MPI: A message passing interface," in *Proceedings of Supercomputing '93*, (Portland, OR), pp. 878–883, IEEE CS Press, Nov. 1993.

[64] M. Ladkau, *A Wide Spectrum Type System for Transformation Theory.* PhD thesis, De Montfort University, 2009.